Università di Genova | DIBRIS DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

MSc Computer Science
Software Security & Engineering - Software Engineering

A serious game for High Performance Computing

Szymon Zinkowicz

Advisor: Prof. Daniele D'Agostino Examiner: Prof. Gianna Reggio

March, 2026

Università di Genova, DIBRIS
Via Opera Pia, 13
16145 Genova, Italy
https://www.dibris.unige.it/

To my father Piotr,
from whom I learned modesty, hard work, and integrity, and whose sacrifices made this possible.

To my mother Wioletta,
who always believed in me and pushed me forward.

Abstract

High Performance Computing (HPC) and parallel programming pose significant pedagogical challenges, as students must internalize abstract concepts without tangible interactive experiences. Serious games facilitate intuitive understanding through engagement, surfacing misconceptions and enabling targeted instruction.

This thesis presents a web-based educational serious game for teaching fundamental parallel computing concepts—specifically OpenMP shared memory parallelism—through interactive card-sorting mechanics. Players sort cards collaboratively in multiplayer mode using local buffer zones, simulating OpenMP thread-local storage. Students cooperate to minimize sorting time under constraints, learning speedup, scalability, communication overhead, and multithreading.

Key contributions include: (1) an original mapping between card-sorting mechanics and HPC concepts derived from physical classroom experiments, (2) robust multiplayer state synchronization using GDSync, (3) responsive UI/UX for manipulating 50+ interactive elements across screen sizes, (4) documentation of technical challenges in GDSync and multiplayer synchronization, and (5) an open-source, extensible platform for HPC education research.

Keywords: Serious Games, High Performance Computing, Parallel Computing Education, OpenMP, Web-Based Game Development, Multiplayer Synchronization, Godot Engine, GDScript, Card Sorting, Educational Technology

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Daniele D'Agostino, for his guidance, valuable insights, and continuous support throughout this thesis work. His much needed enthusiasm was what has put me on the rails for High Performance Computing research. His expertise in High Performance Computing and educational methodologies has been instrumental in shaping this research.

I am also grateful to Professor Maura Cerioli for serving as the reviewer of this thesis for the first period, steering me in right direction regarding organisational aspects and providing constructive feedback.

Special thanks to the Godot Engine community and the developers of the GDSync framework for their open-source contributions and support in resolving technical issues encountered during development.

Finally, I thank my family and friends for their encouragement and patience during the completion of this work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context and Motivation

1.1.1 The Challenge of Teaching High Performance Computing

High Performance Computing (HPC) has become an indispensable tool in modern computational science, powering everything from weather forecasting and molecular dynamics simulations to machine learning and big data analytics. As computational problems grow in scale and complexity, the ability to write efficient parallel programs has transitioned from a specialized skill to a fundamental competency for software engineers and computational scientists.

However, teaching parallel computing concepts presents unique pedagogical challenges. Students frequently find it difficult to visualize how multiple threads or processes interact, communicate, and coordinate to solve problems efficiently. The principal challenges include:

- **Abstraction Gap**: Concepts like threads, processes, synchronization, and message passing are inherently abstract and difficult to visualize
- **Cognitive Load**: Understanding parallel algorithms requires simultaneous consideration of multiple execution flows, shared resources, and synchronization primitives
- **Limited Immediate Feedback**: Traditional HPC assignments involve slow compile-submit-debug cycles that impede learning
- **Language Barriers**: HPC programming typically requires C/C++ or Fortran, languages with steep learning curves that can overshadow conceptual understanding. Students often struggle more with template errors and memory management than with parallelism itself due to the focus of C++ on keeping backwards compatibility as first priority

- **High Entry Barrier**: Setting up HPC environments and debugging distributed systems requires significant technical expertise that distracts from core concepts

Traditional lecture-based approaches struggle to convey the dynamic, interactive nature of parallel processes, motivating the exploration of alternative pedagogical approaches.

1.1.2 Serious Games as Educational Tools

Serious games- Can be defined asgames designed with a primary purpose beyond entertainment - they have emerged as powerful educational tools. By leveraging mechanics such as immediate feedback, progressive challenges, and interactive exploration, serious-game-based tools can make complex concepts more accessible and engaging.

- **Active Learning**: Learners engage directly with concepts through interaction
- **Immediate Feedback**: Actions produce visible outcomes quickly
- **Visualization**: Abstract parallel behaviors become tangible
- **Safe Experimentation**: Risk-free environment for trial and error without costly HPC infrastructure
- **Lower Entry Barrier**: Learners can focus on concepts before full programming complexity

Terminology Note Throughout this thesis, the artifact is referred to as a **game**. For pedagogical precision, it can also be categorized as a **serious game** and used as an instructional tool in educational contexts. This terminology aligns with prior HPC educational activities explicitly framed as serious games [1].

1.1.3 From Physical Experiments to Digital Tooling

The pedagogical approach underlying this thesis stems from classroom experiments conducted by Professor Daniele D'Agostino following the approach presented in *Teaching and learning HPC through serious games* [1], where students physically sorted numbered cards to simulate parallel computing paradigms:

- **Sequential Execution**: A single student sorts all cards alone—demonstrating traditional sequential algorithms

- **OpenMP Simulation**: Multiple students work collaboratively at a shared desk with all cards visible in a common container, using local buffer areas for local sorting—demonstrating shared memory parallelism with partitioned per-thread work areas

These physical activities were highly effective at conveying parallel computing concepts in a tangible, memorable way. However, they faced significant limitations, including scalability (limited by desk size).

This thesis aims to capture that pedagogical effectiveness in a scalable, digital format accessible via web browsers, enabling students worldwide to experience similar learning benefits without the constraints of physical classrooms.

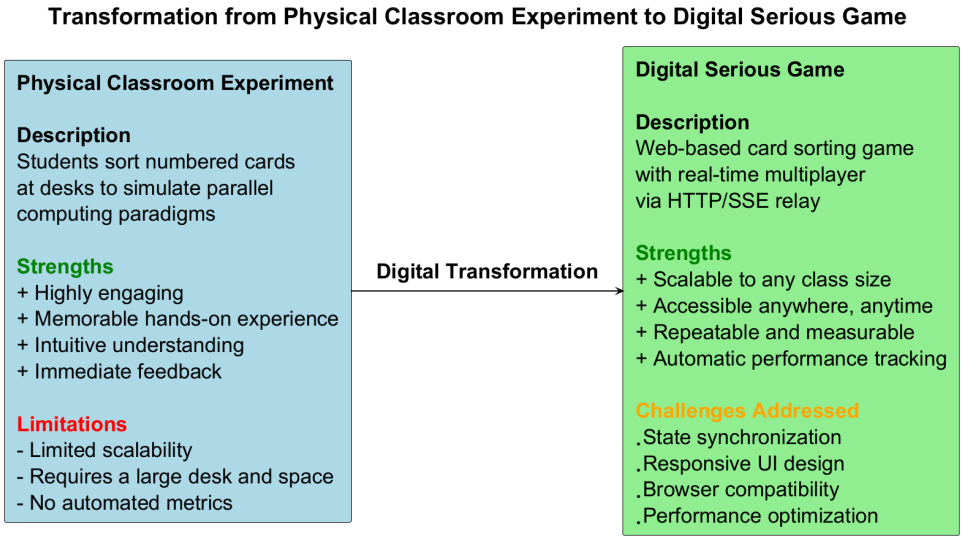


Figure 1.1: Transformation from physical classroom activity to digital serious-game-based educational tool, preserving pedagogical effectiveness while overcoming physical limitations

1.2 Problem Statement

This thesis addresses the following research problem:

> *How can we design and implement an effective web-based serious-game educational tool that teaches fundamental High Performance Computing concepts (specifically OpenMP shared-memory parallelism and sequential vs. parallel execution) through interactive card-sorting while overcoming technical challenges of multiplayer state synchronization and responsive UI/UX constraints?*

This overarching problem decomposes into three interrelated challenges:

- **Educational Challenge**: How can abstract parallel computing concepts be mapped to concrete mechanics that accurately represent OpenMP paradigms?
- **Technical Challenge**: How can real-time multiplayer interaction be implemented across diverse devices with acceptable latency and consistent state?
- **Design Challenge**: How can we balance educational effectiveness with engagement and playability across different screen sizes and devices?

Detailed requirements analysis appears in Chapter 3.

1.3 Research Objectives

The primary objective of this thesis is to develop a functional serious-game-based educational tool that demonstrates the feasibility of teaching HPC concepts through web-based interactive activities, creating an open-source platform suitable for educational use and future research.

1.3.1 Core Objectives

1. **Develop an Interactive Game-like Educational Tool**: Use card sorting as a metaphor for parallel sorting and OpenMP-style collaboration
2. **Implement Web-First Functionality**: Run directly in modern browsers with responsive design
3. **Create Multiplayer Capability**: Support many collaborative participants with synchronized shared state
4. **Evaluate Technical Feasibility**: Assess architecture decisions and implementation challenges, including GDSync integration and HTTP/SSE relay networking
5. **Support Instructor-Guided Use**: Ensure the artifact fits classroom and workshop workflows with facilitation and reflection

1.4 Proposed Solution: HPC Sorting Serious Game

This thesis presents the **HPC Sorting Serious Game**, a web-first serious-game-based educational tool for introducing parallel computing concepts through interactive card sorting. The card interaction model provides an intuitive bridge between concrete manipulation and abstract HPC concepts. Technology selection process, alternatives explored, and final rationale are presented in Chapter 3.

1.5 Thesis Contributions

This thesis contributes to HPC education and educational game engineering in four areas:

1. **Pedagogical Contribution**: A novel game-based approach for teaching OpenMP shared-memory parallelism and sequential vs. parallel execution comparison, directly inspired by successful physical classroom experiments, with systematic mapping between game mechanics and parallel computing paradigms. Additionally, demonstrates that fundamental HPC concepts can be taught effectively without requiring students to first master complex programming languages like C/C++, reducing cognitive load.
2. **Technical Contribution**: A web-first implementation with synchronized multiplayer behavior and responsive UI for high-card-count interaction
3. **Documentation Contribution**: Detailed discussion of practical engineering issues, including synchronization and framework constraints
4. **Open-Source Contribution**: An extensible codebase that can be reused and expanded for future HPC education research

Detailed discussion of these contributions appears in Chapter 8.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 reviews parallel computing fundamentals, serious games in education, web development constraints, and multiplayer architecture patterns.

Chapter 3 describes the research method, requirements analysis, and technology selection.

Chapter 4 presents system architecture, component design, networking model, and UI/UX strategy.

Chapter 5 details implementation choices and key engineering decisions.

Chapter 7 analyzes development challenges and lessons learned.

Chapter 6 presents system capabilities, technical performance, and preliminary educational observations.

Chapter 8 summarizes contributions and outlines future work.

Appendices provide supplementary materials including user documentation and technical references.

Chapter 2

Background and Literature Review

This chapter provides the foundational knowledge and contextual background necessary to understand the HPC Sorting Serious Game. It reviews key concepts in parallel computing, serious games in education, web-based game development, and multiplayer networking architectures.

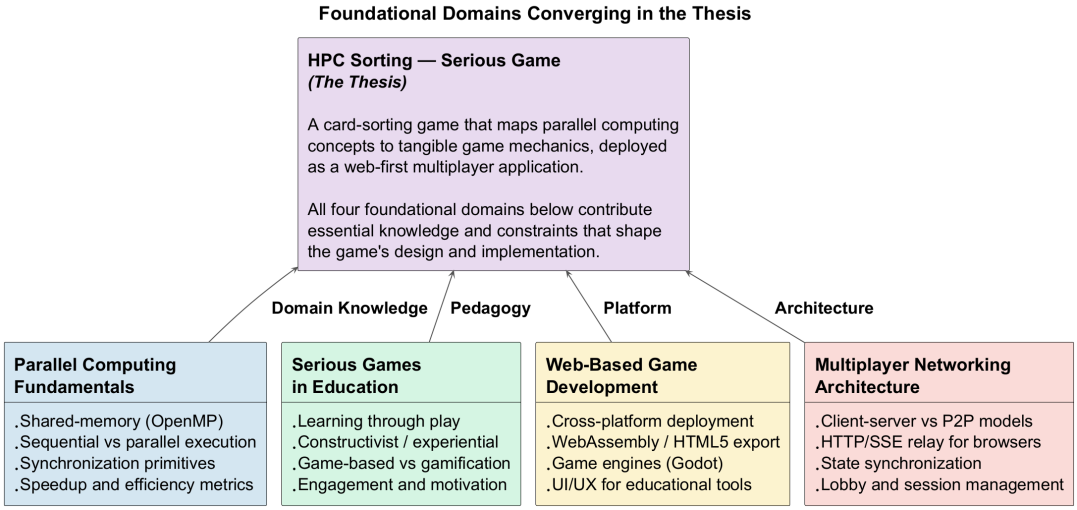


Figure 2.1: Conceptual overview of the four knowledge domains converging in this thesis.

2.1 Parallel Computing Fundamentals

Parallel computing is the simultaneous execution of multiple computational tasks to solve problems more efficiently than sequential processing. As Moore's Law approaches its physical limits, parallel computing has become essential for improving computational performance across scientific computing, data analytics, and real-time systems.

2.1.1 Parallel Programming Paradigms

Two primary paradigms dominate parallel computing education and practice: shared-memory parallelism and distributed-memory parallelism.

Shared-Memory Parallelism

In shared-memory systems, multiple processors or cores access a common address space. All threads can read and write to the same memory locations, enabling efficient data sharing but requiring careful synchronization to prevent race conditions.

Characteristics:

- All processors access the same physical memory
- Communication between threads occurs through shared variables
- Synchronization primitives (locks, barriers, atomic operations) prevent conflicts
- Lower communication overhead compared to distributed memory
- Scalability typically limited to a single workstation (one multi-core node)

Applications: Shared-memory parallelism is ideal for problems requiring frequent communication between processing units, such as iterative algorithms, graph processing, and data analytics on multicore processors.

Distributed-Memory Parallelism

In distributed-memory systems, each processor has its own local memory. Processors communicate by explicitly sending and receiving messages over a network, requiring programmers to manage data distribution and communication patterns.

Characteristics:

- Each processor has private memory space
- Communication between processes requires explicit message passing
- Scalable to thousands or millions of processors
- Higher communication overhead and latency
- No shared state eliminates race conditions but complicates programming

Applications: Distributed-memory parallelism is essential for large-scale scientific simulations, weather modeling, molecular dynamics, and big data processing across cluster and supercomputer architectures.

2.1.2 OpenMP: Shared-Memory Programming

OpenMP (Open Multi-Processing) is a widely-used API for shared-memory parallel programming in C, C++, and Fortran. It uses compiler directives (pragmas) to specify parallel regions, enabling incremental parallelization of sequential code.

Core Concepts:

1. **Parallel Regions**: Code sections executed by multiple threads simultaneously
2. **Work-Sharing Constructs**: Distribute loop iterations or tasks among threads
3. **Data Scoping**: Variables can be shared (visible to all threads) or private (thread-local copies)
4. **Synchronization**: Barriers, critical sections, and atomic operations coordinate thread execution
5. **Scheduling**: Strategies (static, dynamic, guided) control how iterations are assigned to threads

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < N; i++) {
    array[i] = compute_value(i);
}
```

Listing 2.1: Simple OpenMP parallel loop

Educational Value: OpenMP's simplicity makes it excellent for teaching parallel programming concepts. The incremental approach—starting with sequential code and adding pragmas—allows students to observe immediate performance improvements while learning about thread management, data dependencies, and race conditions.

2.1.3 MPI: Distributed-Memory Programming

Example Code:

> **Note:** This section provides background on MPI for completeness and context within the broader HPC landscape. **The game implemented in this thesis focuses exclusively on OpenMP shared-memory parallelism.** MPI distributed-memory concepts are discussed here for educational context but are **not implemented** in the current version. See Chapter 8 for discussion of MPI as potential future work.

MPI (Message Passing Interface) is the de facto standard for distributed-memory parallel programming. It provides a rich set of communication primitives for sending

and receiving messages between processes, enabling scalable parallel computing on clusters and supercomputers.

Core Operations:

Point-to-Point Communication `MPI_Send` and `MPI_Recv` exchange messages between specific processes

Collective Communication Operations like `MPI_Bcast`, `MPI_Gather`, and `MPI_Reduce` involve all processes in a communicator

Process Groups Communicators organize processes and define communication contexts

Data Types MPI supports both primitive and user-defined data types for message content

```
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    // Master process
    MPI_Send(data, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    // Worker process
    MPI_Recv(data, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}

MPI_Finalize();
```

Listing 2.2: Simple MPI message passing

Educational Value: MPI teaches students about distributed computing challenges including data distribution, explicit communication, latency management, and scalability. The message-passing model forces programmers to think carefully about data ownership and communication patterns, skills essential for modern distributed systems.

2.1.4 Parallel Sorting Algorithms

Example Code: Sorting is a fundamental algorithmic problem that benefits significantly from parallelization. Several parallel sorting strategies exist, each with different communication and synchronization requirements.

Parallel Merge Sort:

- Recursively divide array into subarrays
- Sort subarrays in parallel
- Merge sorted subarrays (potentially in parallel)
- Well-suited for shared-memory systems
- Time complexity: $O(\frac{n \log n}{p})$ with p processors

Parallel Quicksort: Parallel quicksort [2] exploits task-level parallelism through its recursive divide-and-conquer structure:

- Select a *pivot* element from the array
- **Partition** the array: elements $\leq$ pivot go left, elements $>$ pivot go right
- Recursively sort the left and right partitions *in parallel* (each assigned to a different thread/task)
- Quality of the pivot determines load balance—a poor pivot creates unequal partitions
- Expected time complexity: $O\left(\frac{n \log n}{p}\right)$ with p processors, assuming balanced pivots

Unlike merge sort, where the *merge* phase is the synchronization bottleneck, quicksort's challenge is the *partition* phase: threads must agree on a pivot and redistribute data before independent recursion can proceed. Parallel partitioning schemes (e.g., each thread partitions a local block, then a prefix-sum determines final positions) can parallelize this step, but at the cost of additional communication [2].

Sample Sort (MPI Pattern):

- Distribute data across processes
- Each process sorts local data
- Select sample elements to determine global pivots
- Redistribute data based on pivots
- Each process sorts its final partition
- Commonly used in distributed-memory systems
- *Note: Not implemented in this thesis; discussed for context*

Relevance to Educational Game: The card-sorting game mechanics map directly to *parallel merge sort* [2]: the game's divide-into-buffers, sort-locally, then merge-back workflow, with barriers separating each merge round, is a faithful model of static-partitioning parallel merge sort. In the implemented game, players act as threads (OpenMP) coordinating through shared memory to sort cards efficiently, making abstract algorithmic concepts tangible through gameplay.

Parallel quicksort, by contrast, requires dynamic pivot selection, runtime repartitioning, and recursive sub-group formation—capabilities that fall outside the game's single-round "pick your interval, sort locally, assemble" mechanics. The game therefore supports **static partitioning strategies** (such as parallel merge sort) but does *not* support **recursive strategies** (such as parallel quicksort). Chapter 4 details the architectural support for merge sort in Section 4.9, and Chapter 8 identifies guided quicksort as a future extension requiring new game mechanics.

2.2 Serious Games in Education

Serious games—games designed primarily for purposes beyond entertainment—have emerged as powerful educational tools. This section reviews the theoretical foundations and empirical evidence supporting game-based learning.

2.2.1 Definition and Characteristics

Zyda [3] defines a serious game as "a mental contest, played with a computer in accordance with specific rules, that uses entertainment to further [...] training, education, health, public policy, and strategic communication objectives." These games leverage game design principles to achieve learning objectives, skill development, or behavioral change.

Key Characteristics:

- **Clear Learning Objectives**: Explicit educational goals aligned with curricula
- **Engaging Mechanics**: Game elements (challenges, rewards, feedback) maintain motivation
- **Active Learning**: Players learn by doing, not passive observation
- **Immediate Feedback**: Instant consequences of actions reinforce learning
- **Safe Experimentation**: Risk-free environment for trial and error
- **Progressive Difficulty**: Challenges scale with player skill development

2.2.2 Theoretical Foundations

Game-based learning draws on several established educational theories:

Constructivism: Constructivist theory [4] posits that learners actively construct knowledge through experience rather than passively receiving information. Games naturally support constructivist learning by placing players in problem-solving scenarios where they discover principles through experimentation.

Flow Theory: Csikszentmihalyi [5] describes "flow" as the optimal psychological state where challenge level matches skill level, producing deep engagement. Well-designed educational games maintain flow by dynamically adjusting difficulty, keeping learners challenged without overwhelming them.

Experiential Learning: Kolb [6]'s experiential learning cycle—concrete experience, reflective observation, abstract conceptualization, active experimentation—aligns naturally with game mechanics. Players experience situations, observe outcomes, form hypotheses, and test them iteratively.

2.2.3 Empirical Evidence

Research consistently demonstrates the effectiveness of serious games for learning:

Connolly et al. [7] Meta-Analysis: A systematic literature review analysing 70 empirical studies found that serious games positively impact learning outcomes across multiple domains. Key findings include:

- Improved knowledge acquisition and retention
- Enhanced perceptual and cognitive skills
- Increased motivation and engagement
- Better affective and motivational outcomes
- Most effective when integrated with traditional instruction

Papastergiou [8] Computer Science Education: A study of digital game-based learning in high school computer science found that students using educational games demonstrated:

- Significantly higher knowledge acquisition

- Greater enjoyment and motivation
- Preference for game-based learning over traditional methods
- Improved attitudes toward computer science

2.2.4 Serious Games in Computer Science

Several successful serious games target computer science education. **CodeCombat** [9] teaches programming through RPG-style gameplay in which players write real Python or JavaScript code to control characters; it is used in thousands of classrooms worldwide. **Lightbot** [10] introduces sequencing, loops, and procedures through a visual puzzle interface that requires no programming syntax, making it accessible to young learners.

Parallel Computing Games: In contrast to introductory programming, very few serious games target advanced topics like parallel computing education. Instructors typically rely on traditional, non-gamified software visualizers or simulation tools to teach parallel algorithm design, highlighting a significant gap in the current game-based learning landscape for advanced computer science topics.

2.2.5 Serious Games for HPC Education

High Performance Computing education faces unique challenges that serious games can address:

Why Games for HPC?

1. **Abstract Concepts**: HPC involves invisible processes (threads, messages, synchronization) that are difficult to visualize through static diagrams
2. **Temporal Dynamics**: Parallelism involves timing, race conditions, and coordination that unfold over time—games naturally model dynamic systems
3. **Scale Mismatch**: Real HPC systems involve millions of operations per second; games can slow down and make these concepts observable
4. **Hands-On Experience**: Students learn parallelism best by experiencing coordination challenges firsthand

Mapping HPC Concepts to Game Mechanics: The following mappings enable intuitive understanding of HPC concepts through gameplay (see Table 2.1).

Table 2.1: HPC concepts mapped to game mechanics

HPC Concept	Game Mechanic
Threads/Processes	Players
Shared Memory	Shared game board/container
Thread-Local Storage	Private player inventory
Synchronization	Turn-taking or barriers
Race Conditions	Conflicting simultaneous actions
Speedup	Time comparison (1 player vs. N players)
Communication Overhead	Time to transfer items between players
Load Balancing	Equal distribution of work

Advantages of Game-Based HPC Learning:

- **No Prerequisites**: Students can experience parallel concepts before learning programming syntax
- **Misconception Surfacing**: Games reveal incorrect mental models (e.g., "more threads always faster")
- **Quantifiable Learning**: Performance metrics (time, moves) provide concrete feedback
- **Collaborative Learning**: Multiplayer games naturally create discussion and peer learning
- **Scalable Delivery**: Web-based games reach students regardless of institutional HPC infrastructure

2.3 Web-Based Game Development

Web browsers provide a universal platform for educational game deployment, eliminating installation barriers and enabling access from any device with a modern browser.

2.3.1 Web Platform Advantages

Accessibility: Web games require no installation—students access the game via URL, eliminating software distribution challenges and platform compatibility concerns.

Cross-Platform: A single web build runs on Windows, macOS, Linux, and even mobile browsers, maximizing reach with minimal development overhead.

Instant Updates: Deploying updates is immediate—no app store approval processes or user-initiated updates required.

Low Barrier: Modern browsers include powerful JavaScript engines, WebGL for graphics, and APIs for real-time communication.

2.3.2 Web Development Challenges

Browser Sandboxing: Security restrictions prevent raw socket access, UDP communication, and local peer discovery—complicating multiplayer networking.

Performance Overhead: WebAssembly and JavaScript have overhead compared to native code, though performance is acceptable for 2D games.

Networking Limitations: Traditional peer-to-peer networking (like WebRTC data channels) can be complex to configure and may not work reliably across all network configurations.

2.3.3 Game Engines for Web

Several game engines support web deployment through HTML5/WebAssembly export, each with different trade-offs relevant to educational game development:

Unity offers a mature ecosystem with extensive documentation and a large asset marketplace [11, 12]. Browser deployment is supported through WebGL builds [13], but web exports tend to be large (often tens of MB), and plan/licensing thresholds can introduce constraints for open-source educational projects [14].

Unreal Engine provides advanced rendering and networking capabilities [15]; however, browser delivery is commonly handled through Pixel Streaming rather than lightweight HTML5/WebAssembly export [16], and its high-fidelity 3D focus is often less suitable for lightweight 2D browser applications.

Godot Engine is fully open-source (MIT license) [17, 18], supports Web export [19], and provides a Python-like scripting language (GDScript) alongside a scene-based architecture oriented toward rapid 2D development.

Cocos2d/Cocos Creator is efficient for 2D applications and supports web publishing alongside cross-platform builds [20, 21], though its tooling ecosystem is smaller and the editor workflow is less integrated than some alternatives.

Key evaluation criteria for an educational web game include export size (affecting load time), 2D rendering quality, scripting accessibility for students who may inspect source code, and licensing compatibility with open educational resources. The engine selection process, evaluation evidence, and final decision are presented in Chapter 3, Section 3.4.2.

2.4 Multiplayer Game Architecture

Multiplayer functionality is essential for simulating OpenMP shared-memory parallelism with multiple players collaborating on shared data. This section reviews common multiplayer architectures and networking technologies.

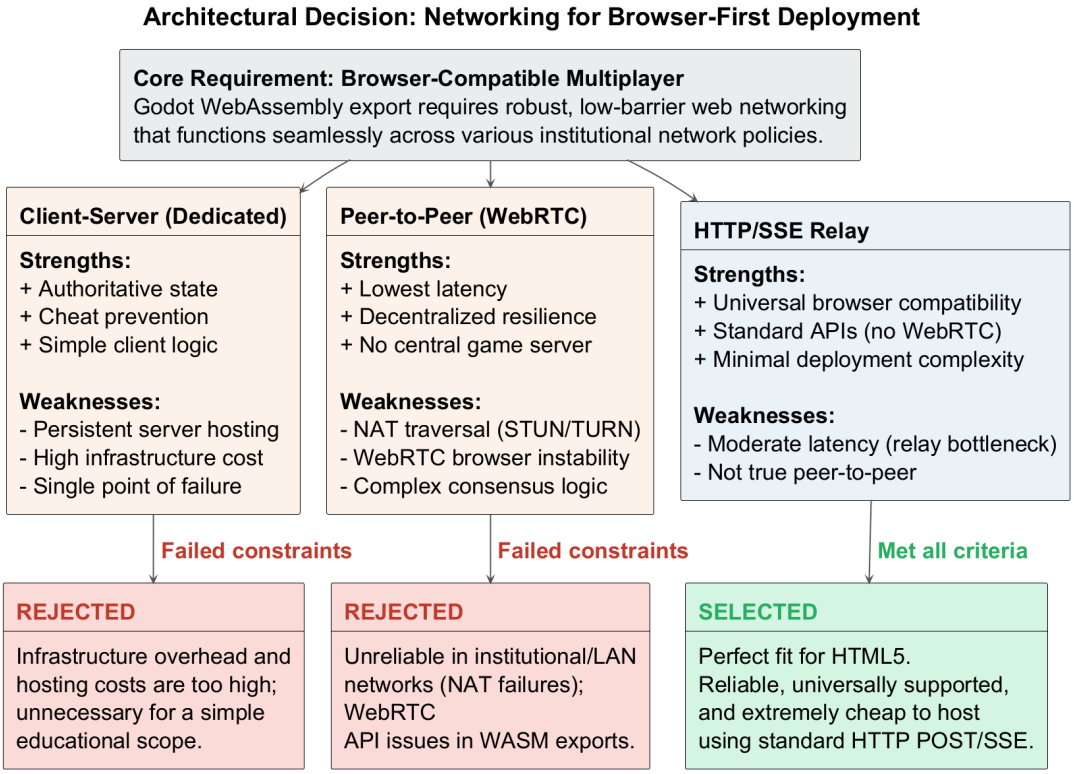


Figure 2.2: Comparison of networking approaches for browser-first deployment, highlighting the HTTP/SSE relay selected for this thesis.

2.4.1 Client-Server Architecture

In a traditional client-server model, a dedicated server hosts the authoritative game state. Clients send their inputs to the server, which validates actions, updates the canonical state, and broadcasts the result to all connected clients. This architecture provides strong consistency guarantees and inherent cheat prevention, since no client can unilaterally alter the game state. However, it requires maintaining dedicated server infrastructure, introduces a single point of failure, and routes all communication through the server, adding latency to every interaction. These costs are justified in scenarios where state correctness is paramount, but they impose operational overhead that may be disproportionate for small-scale educational deployments.

Table 2.2: Summary matrix of networking approach evaluation criteria

Evaluation Criteria	**Client-Server**	**P2P (WebRTC)**	**HTTP/SSE Relay**
Browser compatibility	Partial	Problematic	**Universal**
Infrastructure needs	High	Low (STUN/-TURN)	**Minimal**
Implementation complexity	Medium	High	**Low**
Latency	Low	Lowest	Moderate

2.4.2 Peer-to-Peer Architecture

Peer-to-peer (P2P) networks eliminate the need for a dedicated server by allowing clients to communicate directly. One peer is typically designated as "host" to serve as the authority for conflict resolution. This approach removes server infrastructure costs and can reduce latency for direct connections between nearby peers. However, establishing connections is considerably more complex due to NAT traversal requirements, and the host peer bears a disproportionate performance burden. Cheat prevention is also more difficult, since clients have more control over state. P2P topologies scale well for small groups (2–8 players) but become impractical as group size increases due to the quadratic growth in connection count.

2.4.3 Web Transport Technologies

Web browsers offer several options for real-time communication, each with trade-offs:

HTTP Polling is the simplest approach: the client repeatedly requests updates from the server at fixed intervals. It is universally supported and trivial to implement, but incurs higher latency and server load due to the frequency of requests, making it suitable only for low-frequency state updates.

Server-Sent Events (SSE) allow the server to push events to the client over a persistent HTTP connection. SSE is simpler than WebSocket (it uses standard HTTP with no protocol upgrade), supports automatic reconnection through the browser API, and enjoys broad proxy compatibility. The trade-off is that the channel is unidirectional—server to client only—so client-to-server communication must use a separate mechanism such as HTTP POST.

HTTP + SSE Relay Architecture: For web-based multiplayer games requiring broad browser compatibility, a relay server architecture using HTTP POST for client-to-server communication and SSE for server-to-client events provides:

- Universal browser support without WebSocket complexity
- Works through firewalls and proxies that block WebSocket
- Simplified connection establishment (no STUN/TURN required)
- Trade-off: All traffic routes through server (no P2P)
- Acceptable latency for turn-based or slower-paced games

2.4.4 State Synchronization Patterns

Maintaining consistent game state across multiple clients is challenging. Common patterns include:

Deterministic Lockstep requires all clients to simulate identical game logic, with inputs synchronized and executed simultaneously. This guarantees identical state on every client but is highly sensitive to latency and packet loss, making it most suitable for real-time strategy games (e.g., *Age of Empires*, *StarCraft*).

Client-Side Prediction allows clients to predict the results of their own actions immediately, while the server validates and corrects if necessary. This provides a responsive feel despite latency, at the cost of occasional reconciliation when predictions are wrong. First-person shooters such as *Counter-Strike* and *Overwatch* rely on this technique.

Authoritative Server/Host models maintain a single canonical state on the server or host. Clients send proposed actions and receive authoritative state updates in return. This approach is simpler to implement and reason about, provides stronger consistency, and is well-suited for turn-based or slower-paced games where moderate latency is acceptable.

Suitability for Educational Games: For educational multiplayer games where correctness and pedagogical clarity take priority over competitive low-latency performance, the **authoritative server/host** model is a natural fit: it is straightforward to implement, prevents inconsistencies across clients, and aligns with centralized coordination patterns found in parallel computing. The specific synchronization model adopted for this thesis is discussed in Chapter 3.

2.5 Related Work

This section reviews existing educational tools and games related to parallel computing and serious games.

2.5.1 Parallel Computing Educational Tools

Several tools exist for teaching parallel computing, each with strengths and limitations. Tools such as **Intel VTune** [22] and **TAU** (Tuning and Analysis Utilities) [23] focus on OpenMP performance profiling; while powerful, they have steep learning curves and are designed for optimization work rather than introductory education. **MPICH** [24] and **Open MPI** [25] supply debugging tools for MPI-based programs, but require cluster access or virtual machines and presuppose programming ability.

Gap Identification: Existing tools generally fall into two categories: (1) visualization tools that show parallel execution but lack hands-on interaction, and (2) programming environments that require coding skills, deterring beginners. Few tools bridge the gap between conceptual understanding and practical implementation, and fewer still are designed for web platforms or incorporate game mechanics to enhance engagement.

2.5.2 Serious Games for Computing Education

Several serious games have successfully taught computing concepts, though none specifically target OpenMP-style parallel computing on web platforms. **CodeCombat** [9] teaches programming through RPG gameplay in which players write real Python or JavaScript; it is effective for syntax and basic algorithms but does not address parallelism. **Human Resource Machine** [26] uses visual assembly-like puzzles to teach sequential algorithmic thinking without syntax barriers, but its single-threaded model excludes parallelism concepts. **Shenzhen I/O** [27] targets hardware design and low-level optimization for experienced programmers; while sophisticated, it has no explicit parallel computing focus.

Gap in Parallel Computing Games: No widely-adopted serious game specifically teaches parallel computing concepts like OpenMP shared-memory parallelism through interactive, game-based mechanics on web platforms. This gap motivated the development of the HPC Sorting Serious Game, which focuses on OpenMP concepts with potential for future MPI extensions.

2.5.3 Physical Teaching Methods

The pedagogical approach for this thesis draws inspiration from physical classroom activities:

CS Unplugged: The CS Unplugged [28] collection teaches computer science concepts—algorithms, sorting, searching—through hands-on activities that require no computers. These activities are highly engaging but inherently limited to in-person classroom settings and are difficult to scale or reproduce consistently.

Professor D'Agostino's Card Sorting Experiments: As described in Chapter 1, Professor Daniele D'Agostino conducted classroom experiments where students physically sorted numbered cards to simulate parallel computing paradigms. These experiments demonstrated:

- High student engagement and retention
- Intuitive understanding of parallelism concepts
- Memorable hands-on experience
- Clear differentiation between sequential and parallel execution models

However, physical experiments have limitations:

- Require physical classroom presence
- Not scalable to large classes
- Difficult to reproduce consistently
- Limited flexibility in parameters (card count, student count)
- No performance metrics or automated feedback

Digital Transformation: This thesis aims to capture the pedagogical effectiveness of physical card-sorting experiments in a digital, web-first serious game that overcomes the limitations of physical activities while preserving their educational benefits.

2.6 Summary

This chapter established the foundational knowledge necessary to understand the HPC Sorting Serious Game:

- **Parallel Computing**: Reviewed OpenMP shared-memory parallelism (the focus of this thesis) and MPI distributed-memory paradigms (for context), highlighting their educational challenges and importance.
- **Serious Games**: Examined theoretical foundations, empirical evidence, and successful examples of game-based learning in computer science education.
- **Web Development**: Discussed advantages and challenges of web platforms for educational games, and surveyed available game engines with web export capabilities.

- **Multiplayer Architecture**: Compared client-server and peer-to-peer models, introduced web transport technologies (HTTP/SSE), and justified the host-authoritative approach for OpenMP simulation.
- **Related Work**: Identified gaps in existing parallel computing educational tools and serious games, motivating the development of this project.

The next chapter describes the research methodology, requirements analysis, and technology selection decisions that guided the development of the HPC Sorting Serious Game.

Chapter 3

Methodology

This chapter describes the research methodology, requirements analysis, design approach, and technology selection decisions that guided the development of the HPC Sorting Serious Game. The methodology follows a Design Science Research approach, emphasizing iterative development and evaluation.

3.1 Research Methodology

3.1.1 Design Science Research Framework

This thesis adopts the Design Science Research (DSR) methodology [29, 30], which is particularly well-suited for developing innovative artifacts that address practical problems while contributing to theoretical knowledge.

DSR Process Model: The DSR process consists of six iterative activities:

1. **Problem Identification and Motivation**: Identify the research problem (teaching HPC concepts) and justify the value of a solution (web-based serious game).
2. **Define Objectives of a Solution**: Infer solution objectives from problem definition and existing knowledge (create engaging, accessible, educational multiplayer game).
3. **Design and Development**: Create the artifact (HPC Sorting Serious Game) by selecting appropriate technologies and implementing game mechanics.
4. **Demonstration**: Demonstrate the artifact's utility by applying it to the problem context (students learning parallel computing).
5. **Evaluation**: Observe and measure how well the artifact supports a solution to the problem (usability testing, performance benchmarking).

6. **Communication**: Communicate the problem, artifact, evaluation, and contributions to relevant audiences (thesis, potential publications).

Rationale for DSR: DSR is appropriate for this research because:

- The goal is to create a functional artifact (serious game) rather than merely analyze existing phenomena
- The problem is practical and relevant to HPC education
- The solution requires integrating knowledge from multiple domains (pedagogy, game design, parallel computing, web development)
- Evaluation can be both utility-based (does it work?) and design-based (is it well-constructed?)

3.2 Game Description: The Domain Model

This section provides a complete description of the game as experienced by players. It serves as the *domain model* for the system: all requirements, architectural decisions, and implementation choices in subsequent chapters and sections derive from the gameplay described here. Parallel computing concepts referenced below are introduced in Chapter 2.

3.2.1 Setup

m players (students) are selected to collaboratively sort a deck of n numbered cards. Before play begins the group agrees on a **parallel strategy**—a plan for dividing the work. The canonical strategy is *interval partitioning*: each player is assigned a contiguous block of values. For example, with $n = 100$ cards and $m = 10$ players, player 1 is responsible for cards 1–10, player 2 for cards 11–20, and so on.

The game begins with all n cards placed face-up in a **shared container**—a single space visible and accessible to every player, representing shared memory. A timer starts.

3.2.2 Local Sorting Phase

Each player picks up the cards belonging to their assigned interval from the shared container and moves them into their own **local buffer**—a dedicated work area visible only to that player. Players then sort their subset independently and in parallel. No coordination between players is required during this phase; each player works on their own cards without interference.

3.2.3 Barrier and Assembly Phase

When a player has finished sorting their local subset, they signal readiness by reaching a **barrier point**. No player may proceed to the next step until every player has arrived at the barrier—directly modeling the `#pragma omp barrier` construct from OpenMP. Once the barrier is released, the **main thread player**—the first player to have finished sorting their subset—assembles the individually sorted subsets back into the shared container in the correct global order.

3.2.4 Completion and Measurement

The game ends when all n cards are in the shared container in ascending order. The elapsed time T_m is recorded. Players can then compare this time against a **sequential baseline** T_1—the time measured when a single player sorts all n cards alone in single-player mode—to observe the parallel speedup $S = T_1/T_m$.

Single-player mode. Single-player mode represents sequential execution by a single thread. The player works alone, sorting all cards without any synchronization overhead. The measured time serves as the sequential baseline T_1. A player may also create a multiplayer session without other participants; the gameplay is identical, but operated through the multiplayer interface.

3.2.5 Supported Sorting Strategies

The game's mechanics—pre-assigned fixed intervals, independent local sorting, barrier synchronization, and ordered assembly—naturally support **static partitioning strategies** such as parallel merge sort (divide into buffers, sort locally, merge back). However, the game does *not* support **recursive strategies** such as parallel quicksort. Quicksort requires dynamic pivot selection and recursive repartitioning at runtime: threads must agree on a pivot, redistribute data based on the pivot, and then recursively repeat the process on sub-partitions. This dynamic, multi-round redistribution cannot be expressed through the game's single-round "pick your interval, sort locally, assemble" workflow. The architectural implications of this constraint are discussed in Chapter 4.

3.2.6 Summary of Gameplay Elements

Table 3.1 summarizes the mapping between game elements and their parallel computing counterparts.

Table 3.1: Core gameplay elements and their parallel computing counterparts.

Game element	**HPC concept**	**Mechanic**
m players	m parallel threads	Simultaneous independent action
Shared card container	Shared memory	All players see the same cards
Private player buffer	Thread-local storage	Cards invisible to other players
Interval assignment	Static work distribution	Pre-game agreement
Barrier point	`#pragma omp barrier`	All must arrive before assembly
Main thread player	Master / coordinator thread	First to finish; assembles results
Timer (T_m vs. T_1)	Speedup $S = T_1/T_m$	Measured automatically

3.3 Requirements Analysis

A comprehensive requirements analysis was conducted to ensure the game met educational and functional objectives.

3.3.1 Educational Requirements

The primary purpose of the game is education. Requirements were derived from HPC learning objectives and parallel computing curricula:

ER1: OpenMP Simulation

- The game must simulate shared-memory parallelism
- All players must see all cards in the main container (shared memory)
- Players must have local buffers (partitioned shared memory for per-thread work)
- No explicit communication required between players (implicit synchronization)

ER2: Sequential Execution Baseline

- The game must provide a single-player mode for sequential execution comparison

- A single player must sort all cards without parallelism
- This mode serves as a baseline for understanding speedup from parallelization
- Players should be able to compare sequential vs. parallel performance

ER3: Performance Feedback

- The game must provide timing information (simulating speedup measurement)
- Move counting must encourage algorithmic efficiency
- Visual feedback must reinforce correct and incorrect actions

ER4: Progressive Difficulty

- Support variable numbers of cards (1–200) for scalability lessons
- Support variable numbers of players (E.g. 2–40) for parallelism degree
- Provide different sorting challenges (random, partially sorted, reverse sorted)

ER5: Conceptual Clarity

- Game mechanics must clearly map to HPC concepts
- Visual design must distinguish shared vs. private data
- Terminology should align with parallel computing vocabulary when appropriate

3.3.2 Functional Requirements

Functional requirements specify what the system must do:

FR1: Card Management

- Generate configurable numbers of cards with unique values
- Shuffle cards randomly at game start
- Render cards with clear visual representation (number, state)
- Support dragging and dropping cards between containers

FR2: Single-Player Mode

- Provide a single-player game mode representing sequential execution
- Display all cards in a scrollable container
- Provide local buffer zones for temporary storage
- Validate sorting order upon completion
- Display completion time and move count

FR3: Multiplayer Mode

- Enable many players (e.g. 10–40) to connect via server-relayed networking
- Provide lobby for matchmaking and game configuration
- Display shared container visible to all players (OpenMP shared memory)
- Synchronize card movements across all clients
- Provide local buffers for each player (partitioned shared memory for per-thread sorting)
- Hide cards in other players' local buffers (thread-private data)
- Handle player disconnections gracefully

FR4: User Interface

- Main menu with navigation to single-player and multiplayer modes
- Lobby interface for creating/joining games
- In-game UI with timer, move counter, player indicators
- Victory screen with performance summary
- Settings for theme, controls, and game parameters

FR5: Feedback Systems

- Toast notifications for events (player joined, card moved, errors)
- Visual animations for card pickup, drop, invalid moves
- Sound effects for actions (optional, can be disabled)
- Confetti or celebration effects upon game completion

FR6: Guest Access

- No account creation or authentication required
- Players can join games immediately via a shared lobby code

3.3.3 Non-Functional Requirements

Non-functional requirements specify quality attributes of the system:

NFR1: Usability

- Intuitive interface requiring no tutorial for basic gameplay
- Responsive drag-and-drop with visual feedback
- Support up to 200 cards without noticeable UI lag during event handling

NFR2: Reliability

- Graceful handling of network disconnections
- Game initialization and card generation under 2 seconds

NFR3: Maintainability

- Clean code structure with modular, event-driven design (no per-frame game logic)

3.3.4 Project Constraints

C1: Licensing & Distribution

- Open-source license (MIT) for educational reuse
- Deployable as a static web application without server-side infrastructure for the game itself

3.3.5 Core Concept

The game uses a deck of numbered cards that players must arrange in ascending order. While the task itself is simple, it serves as an analogy for several models of sorting execution used in parallel computing. Different gameplay configurations correspond to different computational paradigms.

Single-Player Mode (Sequential Execution): In this configuration the sorting task is handled by a single participant. All cards are visible and accessible to the player, who performs every step of the ordering process alone. This mirrors the behaviour of a sequential algorithm, where a single process operates on the entire dataset without concurrent assistance.

Multiplayer Mode (OpenMP Shared-Memory Parallelism): The multiplayer configuration introduces cooperative sorting. Multiple players manipulate the same visible card set stored in a shared container, analogous to a shared-memory environment. Each participant may temporarily move subsets of cards into local buffer zones while organizing them before reintegrating them into the common space. Because players act concurrently rather than in strict turns, the interaction resembles the execution model of OpenMP, where threads operate in parallel while implicitly coordinating access to shared data.

A detailed description of the system architecture and the educational rationale behind this mapping is provided in Chapter 4.

3.4 Technology Selection

Technology choices were treated as methodological outcomes of iterative experimentation, not fixed assumptions. This section documents the evaluation process, alternatives explored, observed constraints, and final decisions for both core architecture and the supporting development toolchain.

3.4.1 Tool and Framework Evaluation Process

Selection decisions were made using a consistent set of criteria aligned with thesis goals:

- **Educational fit**: ability to support clear mapping between gameplay mechanics and HPC concepts
- **Web deployment feasibility**: practical browser support and export reliability
- **Multiplayer support**: synchronization model maturity and debugging feasibility
- **Ecosystem maturity**: documentation quality, examples, and community responsiveness
- **Debugging observability**: runtime inspection and logging support
- **Development speed**: iteration time and implementation complexity

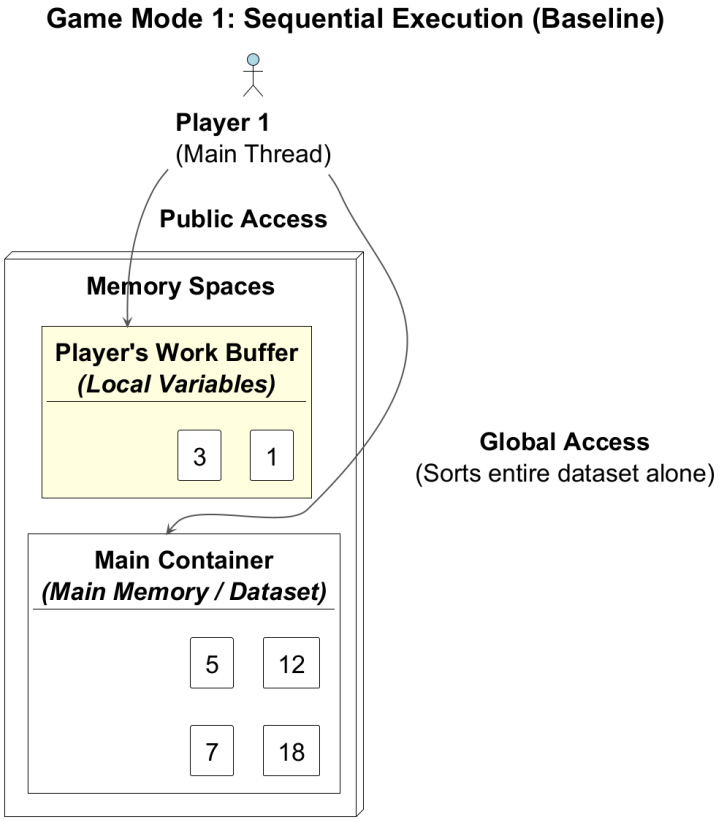


Figure 3.1: Game Mode 1: Sequential Execution (Baseline). A single player (main thread) sorts the entire dataset alone, accessing both the main container (main memory) and a local buffer (local variables).

Pedagogical mapping: A single actor represents a single CPU thread that processes the array element-by-element. No parallel speedup occurs; this mode serves as the baseline (T_1) for calculating parallel efficiency.

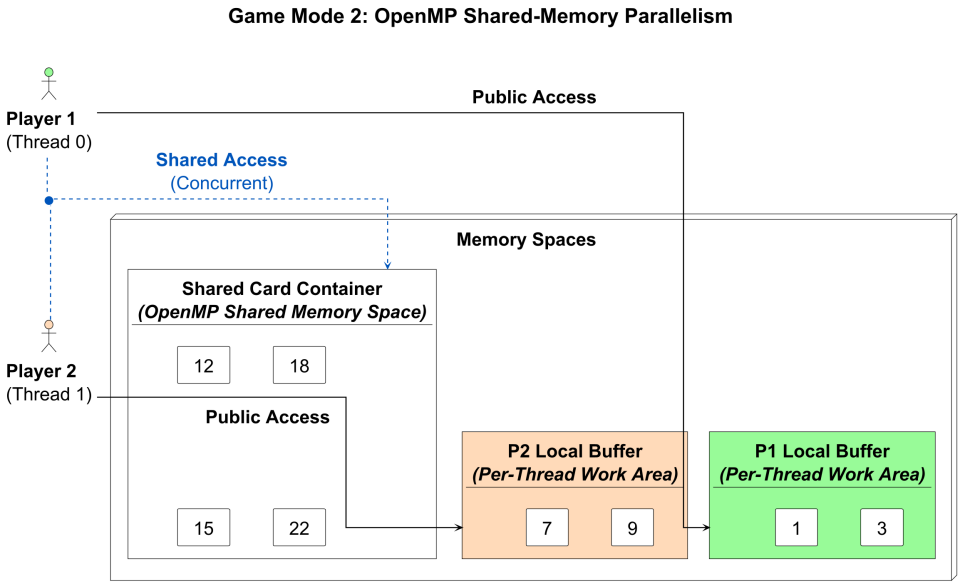


Figure 3.2: Game Mode 2: OpenMP Shared-Memory Parallelism. Multiple players (threads) access a shared card container concurrently and use local buffers for per-thread sorting.

Pedagogical mapping: **Players** represent concurrent threads working on a single task. The **shared container** models a globally accessible memory space, introducing concepts such as race conditions and the need for synchronization (barriers). **Local buffers** represent partitioned regions of shared memory—analogous to per-thread segments of a shared array (e.g., `array[tid*chunk .. (tid+1)*chunk-1]`), where each thread conventionally works on its own portion while the data remains technically accessible to other threads when they reach a barrier. This models common OpenMP patterns such as block-based parallel sorting, where each thread sorts a local chunk before results are merged back into the shared space.

- **Maintainability**: modularity, replacement cost, and long-term project sustainability

Decisions were validated through prototype iterations described in Section 3.4.8, with problematic assumptions revisited as implementation evidence emerged.

3.4.2 Game Engine: Selection

Alternatives explored: Unity, Unreal, Godot, and Cocos2d were evaluated against the criteria in Section 3.4.1.

Practical background and evaluation context: Before this thesis, the author had only brief prior hands-on exposure to Unity and Godot, and no production experience with Unreal or Cocos2d. For this reason, engine selection emphasized thesis-time experimentation and iteration evidence rather than long-term prior specialization.

Unity: was a strong candidate due to its mature ecosystem, large community, and broad platform support. Unity's WebGL export compiles C# game code via IL2CPP to C++, then cross-compiles to WebAssembly using Emscripten [31]. This pipeline produces functional browser builds, but with significant overhead: typical WebGL exports range from 50–100 MB uncompressed, and the generated `UnityLoader.js` bootstrap adds measurable initialization latency before the first frame renders. Unity 6 improved WebGL build sizes somewhat, but exports remain substantially larger than Godot's. Additionally, Unity's WebGL target does not support native multithreading in most browsers (WebWorkers can be used for limited parallelism, but the main game loop remains single-threaded). The free tier displays a mandatory Unity splash screen; removing it requires a Pro license ($2,040/year as of 2025), which is impractical for unfunded educational projects. Unity's asset store and documentation are extensive, but rapid iteration on gameplay experiments was slowed by the C# compilation step and editor reload cycle compared to GDScript's interpreted hot-reload workflow.

Unreal Engine: offers industry-leading rendering and a mature networking subsystem (Unreal Replication). However, Unreal Engine 5 officially removed **HTML5-/WebGL export support** from its platform targets. The Unreal Engine 4 HTML5 export existed as an experimental feature but was never production-stable and produced very large builds (150–300 MB). The current recommended approach for browser access is *Pixel Streaming*, which runs the full game on a GPU-equipped server and streams video frames to the client browser. This architecture requires dedicated server infrastructure per concurrent session—entirely impractical for classroom deployment where many students each need an independent game instance on personal devices. Beyond the web export issue, Unreal's C++ codebase imposes compilation times of several minutes per change, and its editor is resource-intensive

(∼8 GB RAM minimum), making rapid gameplay iteration substantially slower than Godot's sub-second GDScript reload cycle. Unreal's 2D support exists but is not a primary focus of the engine, resulting in additional complexity for what is fundamentally a 2D card interaction game.

Cocos Creator (Cocos2d): Cocos Creator 3.x supports WebGL 2.0 export with reasonable build sizes (15–30 MB), making it the closest competitor to Godot for web-first 2D applications. The engine uses TypeScript/JavaScript as its primary scripting language, which provides good browser affinity. However, Cocos Creator's scene editor, while functional, is less mature than Godot's for rapid educational UI prototyping—particularly for drag-and-drop card interactions, scrollable containers, and dynamic layout management that this project requires. The multiplayer ecosystem is minimal: Cocos Creator provides no built-in high-level networking framework comparable to Godot's `MultiplayerAPI` or third-party solutions like GDSync. Implementing lobby management, state synchronization, and RPC dispatch would require building or integrating external solutions from scratch. Community resources and plugin availability are also more limited, particularly for English-language documentation.

Godot 4.x: provided the best overall balance for this thesis context. Godot 4.x exports to WebAssembly + WebGL 2.0 with a complete game build of approximately 28 MB uncompressed (12 MB gzip-compressed), loading in 3–5 seconds on typical broadband—the smallest export among the four candidates. The engine is fully open-source under the MIT license with no usage fees, splash screen requirements, or revenue caps. Godot's scene-based architecture, where each UI element, card, container, and game mode is a composable scene, naturally supported modular development: single-player and multiplayer modes share the same card and container scenes with mode-specific logic attached via script inheritance. GDScript's interpreted execution enables sub-second iteration cycles (edit → run → observe) without any compilation step. The built-in `MultiplayerAPI` provides `ENetMultiplayerPeer`, `WebSocketMultiplayerPeer`, and `WebRTCMultiplayerPeer` as pluggable transport options, and the third-party GDSync framework (Section 3.4.4) adds high-level lobby and state synchronization on top of these transports.

Table 3.2: Game engine comparison for HPC Sorting Game

Criterion	Unity	Unreal	Godot	Cocos Creator
License	Freemium	5% royalty	MIT (free)	Free
2D Performance	Good	Fair	Excellent	Excellent
Web Export	WebGL	Removed in UE5	WASM	WebGL
Export Size	50–100 MB	N/A (Pixel Streaming)	60–160 MB	15–30 MB
Iteration Speed	Medium (compile)	Slow (C++ compile)	Fast	Medium (transpile)
Learning Curve	Medium	Steep	Gentle	Medium
Networking APIs	Good	Excellent	Good	Minimal
Community Size	Large	Large	Growing	Small

Observed constraints:

- Web target required small export size and predictable browser behavior
- Educational project constraints favored open-source tooling and no licensing risk
- 2D interaction with many UI elements required efficient scene and layout workflows

Decision and justification: Godot Engine was selected because:

1. **Cost**: Completely free and open-source (MIT license), no royalties, subscriptions, or hidden fees. Critical for educational projects with no budget. Unlike Unity's tiered licensing or Unreal's revenue-based royalty, Godot imposes no restrictions regardless of project scope.
2. **2D Optimization**: Godot is specifically optimized for 2D games, unlike Unity and Unreal which prioritize 3D. The 2D rendering pipeline operates independently from the 3D engine, avoiding unnecessary overhead for a card-based interaction game.
3. **Web Export Capabilities**: While Unity WebGL builds can often achieve smaller file sizes due to aggressive managed code stripping and IL2CPP optimizations, Godot provides a frictionless, natively integrated web export process without complex compilation pipelines. Furthermore, a static 135 MB web payload (which is significantly reduced over the network via gzip/Brotli compression) is fundamentally simpler to host than Unreal Engine's heavy Pixel Streaming infrastructure.
4. **GDScript**: Python-like scripting language is accessible to students who may examine the source code. Lower barrier to entry than C# (Unity) or C++ (Unreal/Cocos2D). The interpreted execution model eliminates compilation delays during rapid gameplay iteration.
5. **Scene Architecture**: Godot's composable scene system promotes modular design—card scenes, container scenes, and game mode scenes can be developed and tested independently, then composed into complete game modes. This aligns with software engineering best practices and enabled code reuse between single-player and multiplayer modes.
6. **Built-in Multiplayer**: Godot provides three pluggable transport backends (`ENetMultiplayerPeer`, `WebSocketMultiplayerPeer`, `WebRTCMultiplayerPeer`) through a unified `MultiplayerAPI`. This, combined with the third-party GDSync framework, provided the networking foundation needed for web browser compatibility.
7. **Active Community**: Growing community with extensive tutorials, documentation, and plugins. Version 4.x introduced significant improvements in rendering, scripting, and multiplayer APIs over the 3.x series.

8. **Ethical Alignment**: Open-source philosophy aligns with educational values and ensures long-term availability without corporate dependencies or license changes.

Trade-offs: Godot's limitations include:

- Smaller community and fewer third-party assets compared to Unity
- Fewer commercial games as reference implementations
- Some plugins less mature than Unity Asset Store equivalents
- 3D capabilities lag behind Unity/Unreal (not relevant for this project)

3.4.3 Programming Language: Selection

Alternatives explored: Godot supports three primary programming approaches, each with distinct implications for web export, iteration speed, and maintainability:

GDScript: Godot's native scripting language uses Python-like syntax with static type hints available but not required. Scripts are interpreted at runtime with no compilation step—changes take effect immediately when the scene is re-run. GDScript integrates deeply with Godot's editor through the `@export` annotation system, which exposes script variables as editable properties in the inspector panel. This enabled rapid gameplay parameter tuning (card counts, buffer sizes, timer intervals) without editing code.

C#: Godot 4.x supports C# via the .NET 8 runtime. C# offers stronger static typing, IDE support (IntelliSense, refactoring), and access to the .NET ecosystem. However, the .NET runtime adds approximately 20–30 MB to web export size, and Godot's C# web export pipeline has known compatibility issues: the garbage collector's behavior in WebAssembly can cause unpredictable frame stutters, and threading APIs (`System.Threading`) are restricted in browser sandboxes. Additionally, each code change requires a C# compilation step (∼2–5 seconds), accumulating into measurable delays over hundreds of iteration cycles during development.

C++ (GDExtension): GDExtension allows writing native C++ modules that are compiled as shared libraries and loaded by Godot at runtime. This provides maximum performance but at substantial development cost: each target platform requires its own compilation toolchain (MSVC for Windows, GCC/Clang for Linux, Emscripten for WebAssembly). Cross-compiling a single GDExtension module for web export requires setting up the Emscripten SDK, configuring CMake or SCons build files, and managing ABI compatibility across platforms. This level of build complexity is disproportionate to a 2D card game where scripting performance is never the bottleneck.

Observed constraints:

- Rapid iteration was necessary during repeated gameplay and networking experiments—over the 14-month development period, hundreds of small changes were tested per week
- Implementation needed to remain readable for thesis communication and educational reuse
- Web export size and compatibility needed to remain predictable without additional runtime dependencies
- Cross-platform build complexity needed to remain low (single developer, no CI/CD pipeline for native builds)

Decision and justification: GDScript was chosen because:

- **Integration**: Tight integration with Godot Engine—`@export` annotations, signal connections, and scene tree access are first-class language features rather than API bindings
- **Simplicity**: Python-like syntax is readable and approachable for students who may examine the source code as part of the educational experience
- **Rapid Development**: Dynamic typing and concise syntax accelerate prototyping; the interpreted execution model means zero compilation delay between edits
- **Documentation**: The majority of Godot tutorials, official examples, and community resources use GDScript, reducing the time spent searching for solutions
- **Performance**: Sufficient for 2D game logic; bottlenecks are in rendering (handled by Godot's C++ core) and network latency, not scripting throughput
- **No Web Export Overhead**: Unlike C#, GDScript adds no additional runtime to the web export—the interpreter is already part of Godot's WebAssembly binary

Performance Considerations: GDScript is slower than C++ but acceptable for this use case because:

- Card sorting logic is not computationally intensive
- Godot's rendering engine (written in C++) handles performance-critical operations
- Network latency dominates over local computation time
- Development speed more important than raw execution speed

3.4.4 Multiplayer Framework: Selection

Alternatives explored: Three approaches were considered for multiplayer networking. Each was evaluated not only for basic connectivity, but for the full set of capabilities required by a classroom multiplayer game: lobby management, state synchronization, remote procedure calls (RPCs), security/access control, and browser compatibility.

Option 1: Raw Godot Networking

Godot 4.x provides three built-in transport backends through its `MultiplayerAPI`:

- **`ENetMultiplayerPeer`**: UDP-based transport using the ENet protocol. Provides reliable and unreliable channels with automatic packet ordering. However, ENet uses raw UDP sockets, which are **not available in web browsers**—this backend cannot be used for browser-exported games.
- **`WebSocketMultiplayerPeer`**: TCP-based transport over WebSocket connections. Browser-compatible, but provides no built-in lobby management, matchmaking, or peer discovery. A custom WebSocket server must be implemented to handle room creation, player registration, and message routing.
- **`WebRTCMultiplayerPeer`**: Peer-to-peer transport using WebRTC data channels. Browser-compatible in principle, but requires an external signaling server for connection establishment (SDP offer/answer exchange and ICE candidate relay). Godot provides the peer connection wrapper but no signaling infrastructure.

All three backends expose Godot's low-level `@rpc` annotation system for remote function calls, but none provide higher-level abstractions for state synchronization (automatic variable replication), lobby lifecycle management, player data persistence, host migration, or security whitelisting. Building these features from scratch was estimated to require 4–6 weeks of additional development, reducing time available for educational game design and evaluation.

Option 2: GDSync Framework

GDSync [32] is a third-party Godot addon that provides a complete multiplayer abstraction layer. Understanding its internal architecture was necessary to evaluate whether it could serve as the synchronization backbone for a web-exported educational game.

Internal architecture: GDSync operates in two modes—*cloud* (connecting to GDSync's hosted infrastructure) and *local* (LAN peer-to-peer). The cloud mode connection flow proceeds as follows:

1. **Load balancer discovery**: The client sends an HTTPS request to one of three load balancers (`lb1/lb2/lb3.gd-sync.com`), passing the project's public API key. The load balancer returns a list of available game server IP addresses.

2. **Server selection**: The client pings each returned server via UDP on port 8081, measures round-trip times over five probes, and selects the lowest-latency server.

3. **Transport connection**: On desktop, the client connects via ENet/UDP on port 8080. On web builds (detected via `OS.has_feature("web")`), the framework automatically falls back to `WebSocketMultiplayerPeer` on port 8090.

4. **Encryption**: After connection, the server provides a CBC initialization vector. All subsequent packets are encrypted with AES-256-CBC using the project's private API key, then compressed with zstd before transmission.

Packet system: GDSync uses four logical packet channels: `SETUP` (connection handshake), `SERVER` (lobby operations), `RELIABLE` (guaranteed-delivery game sync), and `UNRELIABLE` (best-effort game sync). Requests are batched per channel, serialized using Godot's native `var_to_bytes()` format, and automatically split when a batch exceeds 20 KB to prevent packet fragmentation.

API surface: The framework exposes high-level functions for lobby lifecycle (`lobby_create()`, `lobby_join()`, `lobby_leave()`), remote procedure calls (`call_func()`, `call_func_on()`), automatic variable synchronization (`sync_var()`, `sync_var_on()`), player management (`player_set_username()`, `player_set_data()`), and ownership tracking (`set_gdsync_owner()`). It also provides a synchronized clock (`get_multiplayer_time()`) using NTP-like five-sample averaging against the host's timer.

Security model: GDSync operates in "protected mode" by default, where only explicitly whitelisted functions, variables, and signals can be accessed remotely. Game code must call `expose_func()`, `expose_var()`, or `expose_node()` for each remotely callable entity. Unregistered RPC calls are silently dropped with no error message—a design that initially caused significant confusion during development (see Chapter 7).

Local mode limitations: GDSync's local multiplayer mode uses UDP broadcast on ports 42354–42373 for lobby discovery and `ENetMultiplayerPeer` on port 8080 for game traffic. Both mechanisms rely on raw UDP sockets, which are **entirely unavailable in browser environments**. The five specific API calls that fail on web are: `PacketPeerUDP.bind()`, UDP broadcast to `255.255.255.255`, `ENetMultiplayerPeer.create_server()`, and `ENetMultiplayerPeer.create_client()`. This incompatibility motivated the HTTP/SSE relay architecture described in Section 3.4.5.

Option 3: Custom WebRTC Protocol

Building a custom WebRTC-based multiplayer system from scratch would provide maximum control but requires implementing multiple complex subsystems:

Signaling server: WebRTC connections cannot be established without a signaling server to exchange Session Description Protocol (SDP) offers and answers between peers. This server must handle room creation (generating unique room codes), peer registration, and bidirectional message relay during the connection handshake phase. A WebSocket-based server is the standard approach, requiring implementation of connection lifecycle management, room state tracking, and graceful cleanup when peers disconnect.

ICE/STUN/TURN infrastructure: Each peer must create an `RTCPeerConnection` configured with at least one STUN server (e.g., `stun:stun.l.google.com:19302`) for NAT traversal. STUN enables peers behind standard NATs to discover their public IP address and port mapping. However, peers behind symmetric NATs or restrictive firewalls—common in university and corporate networks where this game would be deployed—cannot establish direct connections via STUN alone. In such cases, a TURN relay server is required to forward all media traffic, adding infrastructure cost (bandwidth charges per session) and latency (extra network hop). Configuring and testing NAT traversal across diverse classroom network environments would add substantial debugging effort.

Data channels and game protocol: Once a WebRTC connection is established, `RTCDataChannel` provides both reliable (TCP-like) and unreliable (UDP-like) message passing. However, the game protocol layer must still be implemented: packet serialization format, message type dispatch, request batching, ordering guarantees, and bandwidth management. None of this is provided by WebRTC itself.

Application-level features: Beyond the transport layer, all application-level features must be built from scratch: lobby management (room creation, player lists, join/leave lifecycle), host election and migration, state synchronization (variable replication, conflict resolution), RPC dispatch (function exposure, parameter serialization), ownership tracking, and a security model to prevent unauthorized remote calls. These features alone represent weeks of development effort.

Partial solution—PackRTC addon: The PackRTC addon [33] (already present in this project's `addons/` directory) provides a partial implementation: it handles WebRTC signaling via a hosted server at `packcloud.himaji.xyz`, supports both mesh and server/client topologies, and manages ICE candidate exchange. However, PackRTC covers *only* the transport layer—it provides no lobby management, state synchronization, RPC system, or security model. Building these features on top

of PackRTC was estimated to require 6–8 weeks, comparable to the raw Godot networking approach.

Estimated total effort: 8–12 weeks for a complete implementation, with significant additional debugging overhead for NAT traversal edge cases across diverse classroom network configurations.

Table 3.3: Multiplayer framework comparison

Capability	**Raw Godot**	**GDSync**	**Custom WebRTC**
Lobby management	Not provided	Built-in	Must implement
State synchronization	Not provided	Automatic variable sync	Must implement
RPC system	Low-level `@rpc`	High-level `call_func`	Must implement
Security model	Not provided	Whitelist-based	Must implement
Web browser support	WebSocket only	WebSocket fallback (cloud)	Native WebRTC
External dependency	None	GDSync servers or fork	Signaling + STUN/TURN
Estimated effort	4–6 weeks extra	Usable immediately	8–12 weeks extra

Observed constraints:

- Multiplayer state complexity required reusable abstractions, not only low-level RPC wiring—the game synchronizes card positions, buffer contents, player data, barrier states, and move counters across all clients
- Browser deployment exposed transport limitations that ruled out ENet-based and UDP-based approaches entirely
- Debuggability and incremental adoption were critical; the framework needed to work for basic use cases while allowing progressive customization
- The 14-month thesis timeline did not permit 8–12 weeks of networking infrastructure development before game design work could begin

Decision and justification: GDSync was selected as the multiplayer framework because it was the only option that provided lobby management, state synchronization, RPC dispatch, and a security model out of the box, enabling immediate

focus on game logic rather than networking infrastructure. Its cloud mode WebSocket fallback provided a path to browser compatibility, and its high-level API (`call_func()`, `sync_var()`, `lobby_create()`) reduced the multiplayer boilerplate code by an estimated 60–70% compared to raw Godot networking.

Challenges encountered: GDSync introduced several difficulties that required workarounds:

- **Protected mode**: The default security model silently drops unregistered RPC calls with no error message or log entry. This caused hours of debugging before the root cause was identified. The workaround required explicit `expose_func()` calls for all 12 remotely callable functions during multiplayer initialization (see Chapter 7).
- **Documentation gaps**: Several API behaviors (packet channel selection, ownership transfer semantics, reconnection lifecycle) were underdocumented, requiring source code inspection of the framework to understand actual behavior.
- **Opaque error handling**: Framework errors surfaced as generic connection failure codes without indicating which specific operation failed or why.
- **Local mode web incompatibility**: As described above, the local multiplayer mode is entirely non-functional in browser environments, necessitating the HTTP/SSE relay architecture (Section 3.4.5).

Despite these challenges, GDSync provided net benefits by enabling functional multiplayer within the first week of integration rather than after months of infrastructure development. During development, a framework issue affecting web export behavior required a local fork maintained by the author [34]. A pull request was submitted to the upstream GDSync repository [32], but until that change is merged and released, this project uses the author's fork for the web-export-based game build.

3.4.5 Networking Technology: Selection

The transport layer decision was driven by the fundamental constraint that GDSync's local multiplayer mode is entirely non-functional in browser environments (Section 3.4.4). A browser-compatible transport was required that could replace GDSync's UDP/ENet local server while preserving its high-level API contract.

Alternatives explored:

WebRTC (peer-to-peer)

WebRTC is the browser-native peer-to-peer protocol and would appear to be the natural choice. However, several practical constraints made it unsuitable as the primary transport:

- **Signaling requirement**: WebRTC connections require an external signaling server to exchange SDP offers/answers and ICE candidates before any peer-to-peer data can flow. This server must remain available for the duration of connection establishment.

- **NAT traversal**: Classroom and university networks frequently use symmetric NATs or restrictive firewalls. In these environments, STUN alone is insufficient—a TURN relay server would be needed, which reintroduces a centralized server (negating the peer-to-peer advantage) while adding bandwidth costs per session.

- **GDSync integration**: GDSync's local server module does not support WebRTC at all—it uses ENet/UDP exclusively. Replacing it with WebRTC would require restructuring GDSync's internal `LocalServer` class, which manages lobby state, client tracking, and packet routing. This was estimated at 3–4 weeks of framework-level work, with risk of introducing regressions in the complex networking stack.

- **Mixed content blocking**: When the game is served via HTTPS (e.g., GitHub Pages), browsers block WebSocket connections to `ws://` (non-TLS) signaling servers. The signaling server would need a valid TLS certificate and public domain, complicating classroom deployment where instructors typically run a server on the local network.

The PackRTC addon available in the project provides WebRTC via a hosted signaling server (`packcloud.himaji.xyz`), but relying on a third-party hosted service for classroom use introduces availability risk and requires internet connectivity even for LAN-only sessions.

Raw Godot networking (ENet/WebSocket)

Godot's built-in `WebSocketMultiplayerPeer` is browser-compatible, but provides only the transport layer. All lobby management, peer tracking, message routing, and GDSync API compatibility would need to be built on top—essentially reimplementing GDSync's `LocalServer` logic with a WebSocket backend. This approach was considered but rejected in favor of HTTP/SSE, which offered simpler debugging (standard HTTP tools), no persistent connection management complexity, and compatibility with standard web infrastructure (reverse proxies, load balancers, CORS).

Decision and justification: HTTP + Server-Sent Events (SSE) To overcome browser limitations while maintaining GDSync API compatibility, a custom relay server architecture was developed using standard web protocols:

1. **HTTP POST for commands**: All game packets (card moves, barrier signals, lobby actions) are sent as standard HTTP POST requests with JSON-encoded bodies. This leverages the browser's native `fetch()` API, which is universally supported, works over both HTTP and HTTPS, and passes through firewalls and proxies without special configuration.
2. **Server-Sent Events (SSE) for real-time push**: Each client opens a persistent `GET` connection to the server's event stream endpoint. The server pushes game events, lobby updates, and peer notifications as SSE text frames. SSE is a W3C standard supported by all modern browsers, uses standard HTTP (no protocol upgrade), and automatically reconnects on connection loss.
3. **Server relay (star topology)**: All communication passes through the relay server rather than flowing peer-to-peer. Clients send packets to the server, which routes them to the appropriate recipients based on lobby membership. The designated host client receives, processes, and broadcasts game state through the server.

Relay server implementation: The relay server is implemented in Python 3.11+ using the aiohttp asynchronous web framework [35, 36]. It maintains in-memory lobby state (no database required) and provides the following REST API:

- `POST /api/lobby/connect` — Client registration: assigns a unique peer ID and returns server configuration
- `POST /api/lobby/create` — Creates a new lobby with a generated room code, registers the creator as host
- `POST /api/lobby/join` — Joins an existing lobby by room code, notifies existing members via SSE
- `POST /api/lobby/broadcast` — Accepts a GDSync-format game packet and relays it to all other lobby members via their SSE streams
- `GET /api/lobby/events` — Opens an SSE stream for the client; the server pushes `game_packet`, `peer_joined`, `peer_left`, and `lobby_closed` events through this channel
- `GET /api/server/info` — Health check and connectivity validation endpoint

The SSE stream includes periodic keep-alive heartbeats to prevent connection timeout by intermediate proxies. Game packets are tunneled as opaque byte arrays—the relay server does not parse or validate game state, maintaining a clean separation between transport and game logic.

Browser security constraints: Two additional browser security mechanisms affected the deployment model:

- **Mixed content blocking**: Browsers serving the game from an HTTPS origin (e.g., GitHub Pages) block all requests to HTTP endpoints. This means a relay server running on a local network at `http://192.168.x.x:3000` cannot be reached from an HTTPS-hosted game. The deployment model must use either full HTTPS (with a valid TLS certificate on the relay server, enabled by mkcert for local development) or full HTTP (both game and server on the local network).
- **Private Network Access**: Modern browsers implement the Private Network Access specification, which blocks requests from public origins to private network addresses (`192.168.x.x`, `10.x.x.x`) even with permissive CORS headers. This further constrains the deployment topology to either full-LAN or full-public setups.

These constraints and their workarounds are documented in detail in the project's deployment guide.

Latency analysis: The HTTP/SSE relay introduces additional latency compared to direct peer-to-peer connections:

- **LAN deployment**: HTTP POST round-trip adds approximately 5–20 ms (single network hop to relay server). SSE event delivery adds <5 ms from server to client. Total additional latency: ∼10–25 ms per game action.
- **Internet deployment**: HTTP POST round-trip adds approximately 50–150 ms depending on geographic distance. SSE delivery adds comparable latency. Total additional latency: ∼100–300 ms per game action.
- **Acceptability**: For a card-sorting game where actions occur at human interaction speed (one card move every 1–3 seconds), sub-second synchronization is sufficient. The game is not latency-sensitive in the way that real-time action or FPS games are. Players do not perceive delays under 300 ms as disruptive to gameplay flow.

3.4.6 Supporting Development Toolchain

Beyond core engine/networking choices, several tools were selected to address specific development challenges encountered during iterative prototyping. Each tool was adopted to solve an observed problem rather than selected speculatively.

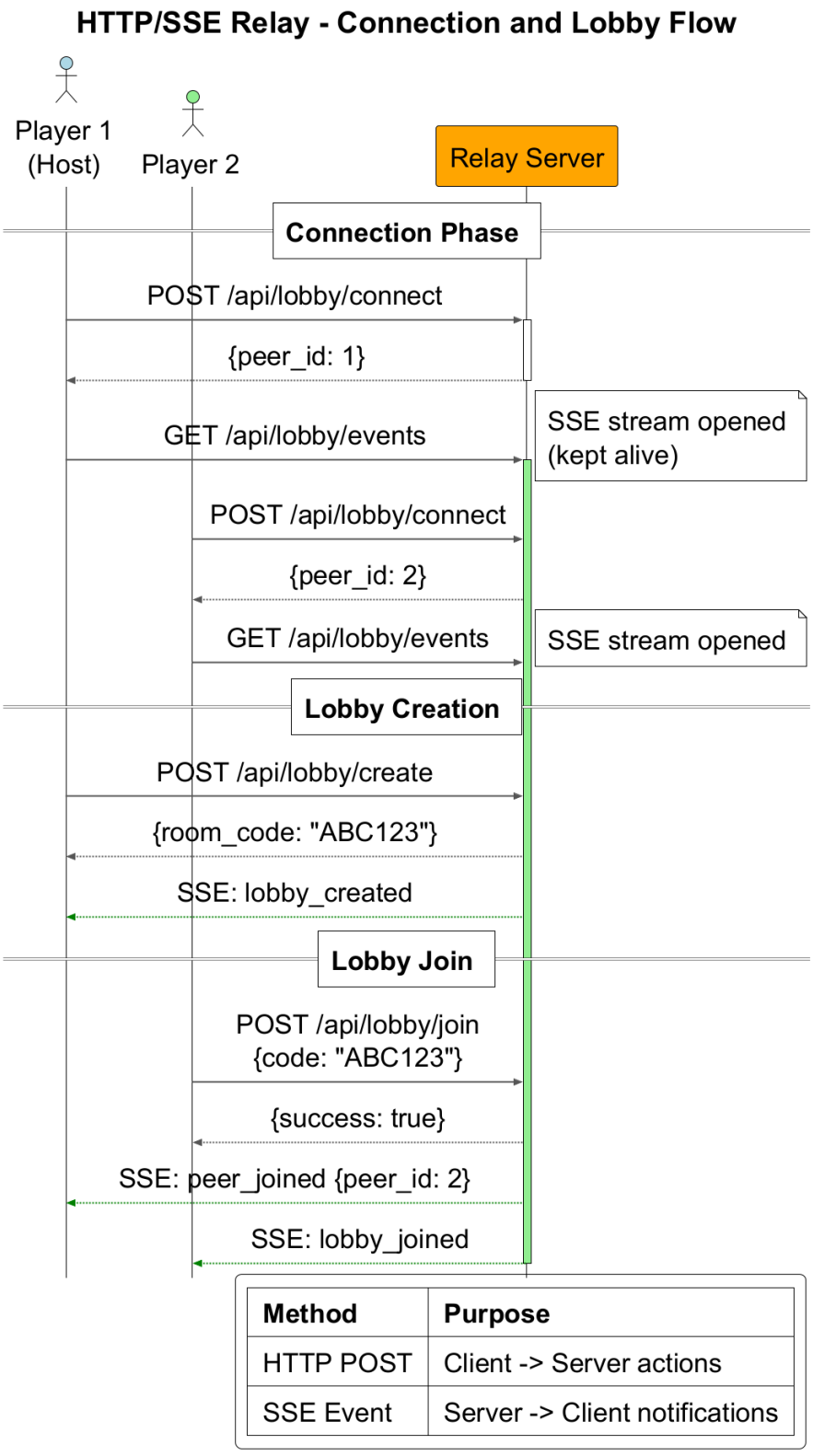


Figure 3.3: HTTP/SSE relay setup flow: client connection, SSE subscription, lobby creation, and lobby join

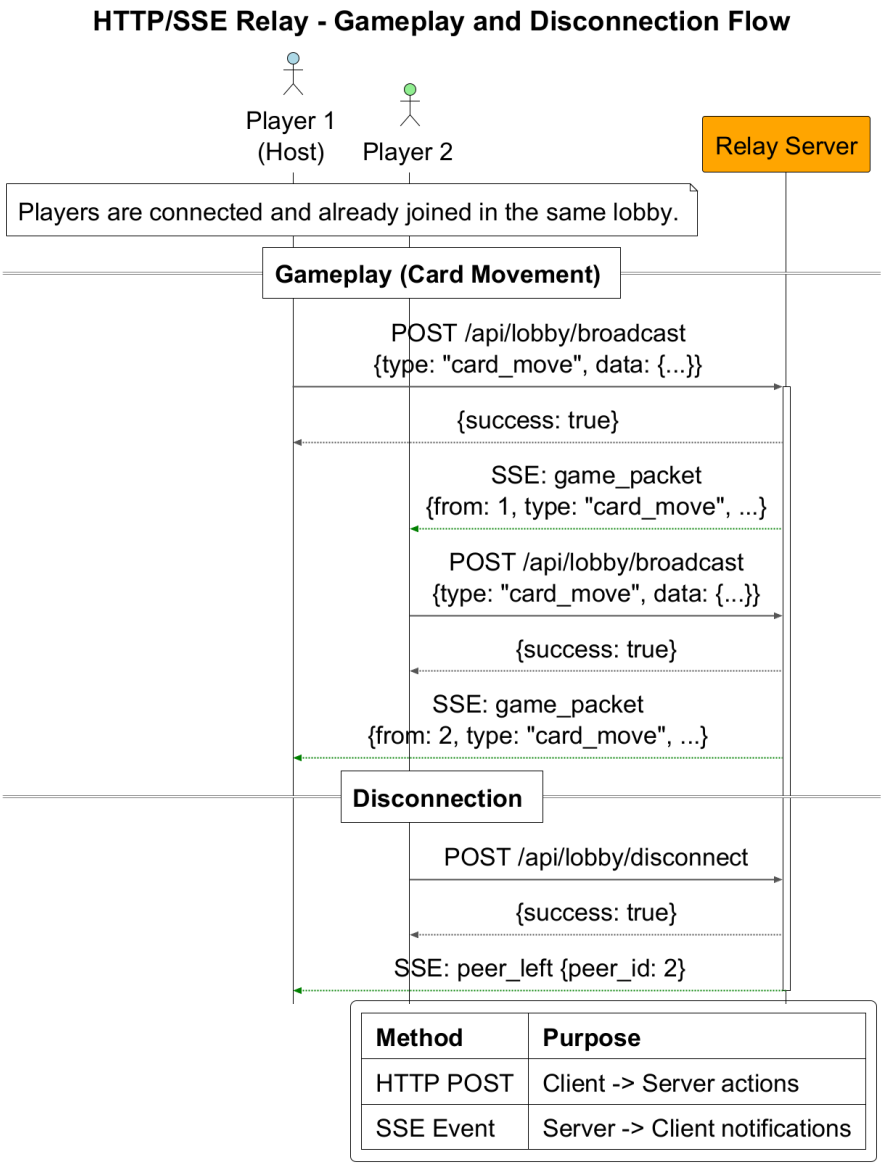


Figure 3.4: HTTP/SSE relay runtime flow: gameplay packet broadcast and disconnection handling

- **ToastParty (Godot plugin)**: Provides non-blocking toast notification popups within the game UI. Used to display multiplayer events (player joined, player left, card conflicts, connection status) without interrupting gameplay flow. Without this, event feedback required checking debug logs or watching the console, which is impractical during live multiplayer testing sessions [37].

- **VarTree (Godot plugin)**: Displays a real-time tree view of all synchronized variables and their current values during runtime. This was essential for debugging state synchronization issues—when a card appeared in the wrong position on one client, VarTree immediately showed which variable was out of sync without adding temporary `print()` statements [38].

- **Custom logging library**: A structured logging system that timestamps and categorizes network events, gameplay actions, and synchronization operations into separate log channels. This enabled post-session analysis of timing issues (e.g., determining whether a card move arrived before or after a barrier signal, on which client) that could not be diagnosed from real-time observation alone.

- **Python/aiohttp relay server**: The HTTP/SSE relay backend described in Section 3.4.5. Implemented in Python 3.11+ with the aiohttp async framework because Python's ecosystem provided the fastest path to a working relay server—async HTTP handling, SSE support, and JSON serialization are all available as mature, well-documented libraries [35, 36].

- **Ruff**: A fast Python linter and formatter that enforces consistent style across the relay server codebase. Adopted because the server code was modified frequently during protocol experiments, and automated style enforcement prevented formatting drift [39].

- **justfile**: A command runner (similar to `make` but language-agnostic) that encapsulates common development workflows as named tasks: building the web export, starting the relay server, launching multiplayer test sessions, running linters, and generating TLS certificates. This replaced a growing collection of ad-hoc shell commands and ensured reproducible workflows across Windows and Linux development environments [40].

- **mkcert**: Generates locally-trusted TLS certificates without requiring a public Certificate Authority. This was necessary because modern browsers require a Secure Context (HTTPS) for many Web APIs, and mixed-content policies block HTTP requests from HTTPS-served pages (Section 3.4.5). During local development, mkcert enabled the relay server to run with a valid TLS certificate trusted by the developer's browser, allowing end-to-end HTTPS testing identical to the production deployment [41].

- **Chocolatey**: A Windows package manager used to standardize the development environment setup (installing Python, Node.js, and other dependencies). Reduced the "it works on my machine" risk when switching between development workstations [42].

- **const_generator (Godot plugin)**: Automatically generates a structured set of GDScript constants for project resources (scene/file paths, node names, input actions). This removed hard-coded string paths across scripts, so renaming files or scenes did not break links—generated constants updated with the new names and prevented stale-reference bugs [43].

- **Git + GitHub**: Version control with branching for experimental features (networking experiments, UI prototypes) and issue tracking for bug management. GitHub also hosted project documentation and enabled collaborative review with the thesis supervisor [44, 45].

3.4.7 Final Toolchain and Rationale Summary

Table 3.4: Final toolchain decisions and accepted trade-offs

Tool	Category	Selected because	Trade-off accepted
Godot 4.x	Engine	28 MB web exports (WASM+WebGL2), MIT license, native 2D pipeline, sub-second iteration	Smaller ecosystem and fewer third-party assets than Unity/Unreal
GDScript	Language	Interpreted (no compile step), Python-like readability, no web export size overhead	Lower peak performance than C++ or C#
GDSync	Multiplayer framework	Built-in lobby, state sync, RPC, and security model; functional in first week of integration	Protected mode debugging overhead, local mode web-incompatible, framework fork required
HTTP/SSE relay (Python)	Transport	Universal browser compatibility, standard HTTP tooling, 10–25 ms LAN latency	Requires relay server infrastructure; not peer-to-peer
Supporting plugins/utilities	Toolchain	Solved specific debugging (VarTree), feedback (ToastParty), and deployment (mkcert) problems	Additional dependency maintenance surface

3.4.8 Development Approach

The game development followed an iterative, incremental approach spanning approximately 14 months (December 2024 to February 2026). Figure 3.5 shows the five phases, key milestones, and decision pivots that shaped the architecture and tooling (see also Sections 3.4.1 and 3.4.7; Chapter 7, Section 7.3.3).

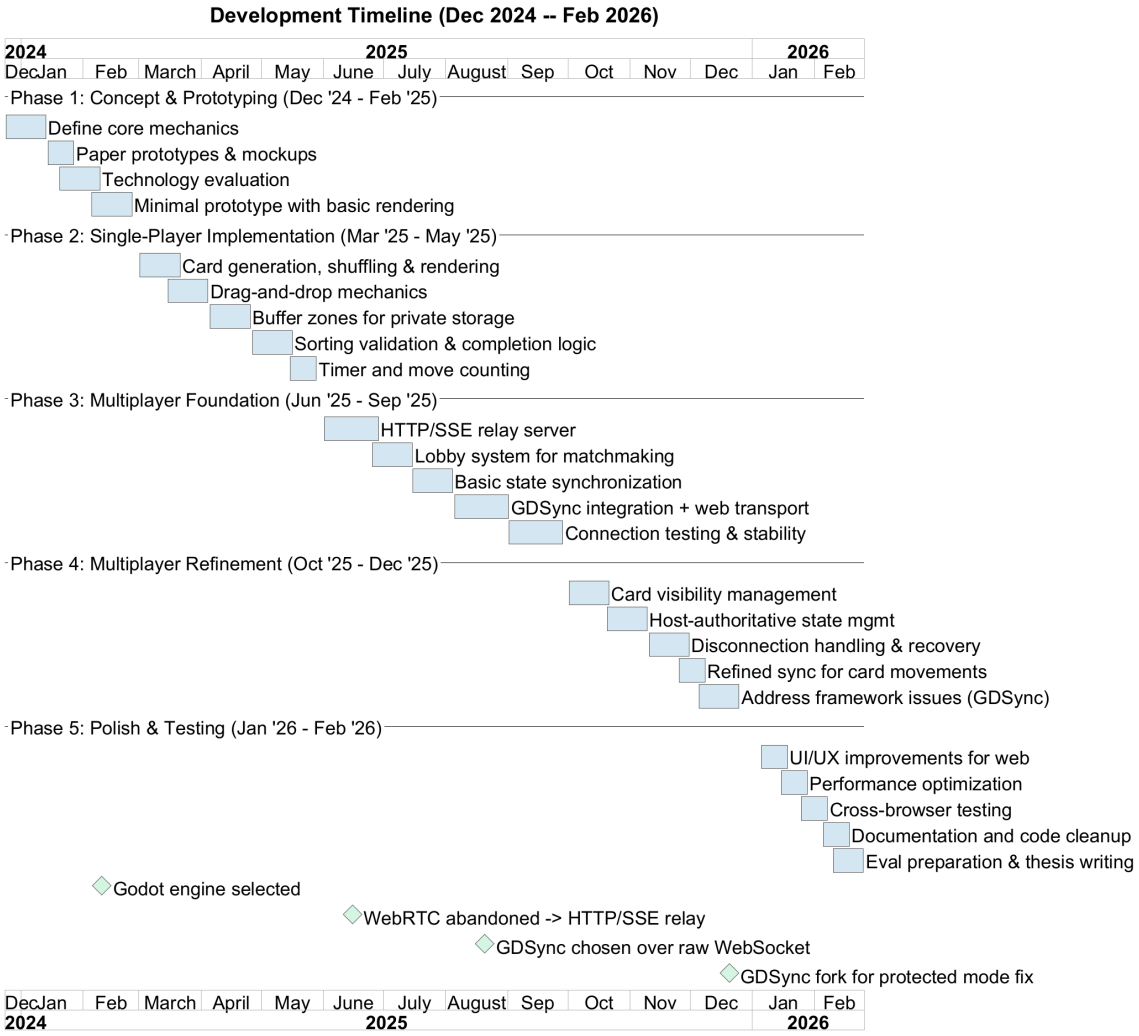


Figure 3.5: Development timeline showing five project phases (December 2024 – February 2026) with key decision pivots marked as milestones.

Phase 1: Concept and Prototyping (December 2024 – February 2025)

- Defined core mechanics based on physical card-sorting experiments
- Created paper prototypes and mockups
- Evaluated technology options (game engines, frameworks)
- Developed minimal prototype with basic card rendering

Phase 2: Single-Player Implementation (March – May 2025)

- Implemented card generation, shuffling, and rendering
- Developed drag-and-drop mechanics for touch and mouse input
- Created buffer zones for private card storage
- Implemented sorting validation and game completion logic
- Added timer and move counting

Phase 3: Multiplayer Foundation (June – September 2025)

- Developed HTTP/SSE relay server for web browser compatibility
- Implemented lobby system for matchmaking
- Developed basic state synchronization
- Integrated GDSync framework with custom web transport layer
- Tested connection establishment and stability

Phase 4: Multiplayer Refinement (October – December 2025)

- Implemented card visibility management (local buffers)
- Developed host-authoritative state management
- Added disconnection handling and recovery
- Refined synchronization for card movements
- Addressed framework issues (GDSync protected mode)

Phase 5: Polish and Testing (January – February 2026)

- UI/UX improvements for web usability
- Performance optimization (rendering, networking)
- Extensive testing across web browsers
- Documentation and code cleanup
- Preparation for evaluation and thesis writing

3.5 Game Design Methodology

3.5.1 Mapping HPC Concepts to Game Mechanics

The core challenge was translating abstract parallel computing concepts into tangible game mechanics. Table 3.5 shows the systematic mapping process:

Table 3.5: Systematic mapping of HPC concepts to game mechanics

HPC Concept	**Real-World Analogy**	**Game Mechanic**
Sequential Execution	Single student working alone	Single player in solo mode
Thread (OpenMP)	Student with access to shared desk	Player in multiplayer mode
Shared Memory	Desk visible to all students	Shared card container
Per-Thread Work Area	Student's section of a shared desk	Player's local buffer zones
Parallel Execution	Students work simultaneously	Multiple players, no turn-taking
Synchronization	Students coordinate access	Shared container access patterns
Memory Access Overhead	Time to reach shared desk	Network latency to shared data
Speedup	Multiple students finish faster	Timer comparison: solo vs. multi
Load Balancing	Even distribution of work	Fair work distribution among players

Design Validation: The pedagogical mapping was validated through:

- Comparison with Professor D'Agostino's physical experiments
- Review by Professor D'Agostino for conceptual accuracy
- Iterative refinement based on feedback

3.5.2 User Experience Design

Design Principles:

1. **Minimize Cognitive Load**: Simple, clear visuals; avoid information overload
2. **Immediate Feedback**: Every action produces visual/audio response
3. **Progressive Disclosure**: Show information when needed, hide complexity initially
4. **Error Prevention**: Design interface to prevent common mistakes
5. **Aesthetic Simplicity**: Clean, uncluttered design focuses attention on cards

Interaction Design: Card interactions were designed to work with both touch and mouse input:

- **Click/Tap to Select**: Quick click or tap selects a card (alternative to drag)
- **Drag to Move**: Click-and-hold initiates drag, release drops card
- **Scroll**: Vertical scroll moves through card container
- **Visual Feedback**: Cards scale up when dragged, drop zones highlight

Color Coding: Colors convey semantic meaning:

- **Blue**: Normal cards, main container
- **Yellow**: Highlighted cards during drag
- **Gray**: Disabled/hidden cards (other players' buffers)
- **Purple**: Player-specific UI elements (buffers, indicators)

3.6 Evaluation Methodology

3.6.1 Evaluation Criteria

The educational tool's success was evaluated across four primary dimensions:

1. **Technical Performance**: Frame rate stability, memory consumption, and cross-browser compatibility.
2. **Functional Completeness**: The successful implementation of all core required features.
3. **Usability**: General ease of use, user learnability, and overall satisfaction.
4. **Educational Effectiveness**: Indicators of user engagement and preliminary conceptual understanding (evaluated informally).

3.6.2 Evaluation Methods

To assess the criteria outlined above, the following methodologies were applied:

Technical Benchmarking

- Performance profiling utilizing Godot's built-in diagnostic tools.
- Frame rate monitoring across target web browsers.
- Memory usage tracking during extended gameplay sessions.

Usability Testing

- Think-aloud protocols conducted with test users.
- Direct observation of first-time players interacting with the system.
- Measurement of task completion rates and completion times.
- Administration of the System Usability Scale (SUS) questionnaire to quantify user satisfaction.

Educational Assessment

- Pre- and post-test knowledge assessments to measure conceptual growth.
- Concept mapping to evaluate the formation of parallel computing mental models.
- Qualitative feedback gathering regarding the perceived learning value of the serious game.

3.7 Development Tools and Environment

Development Platform:

- **OS**: Windows 11
- **IDE**: Visual Studio Code with GDScript extension
- **Engine**: Godot Engine 4.5.x
- **Version Control**: Git with GitHub repository

Testing Environment:

- Primary: Chrome 145+ (Windows, Linux)
- Secondary: Firefox 148+ (Windows, Linux)
- Tertiary: Edge 145+ (Linux)
- Local development server for testing

Build and Deployment:

- Web export (HTML5) built via Godot export templates
- Debug builds for development testing
- Release builds with optimizations for distribution
- Static file hosting for deployment

3.8 Summary

This chapter described the comprehensive methodology employed to develop the HPC Sorting Serious Game:

- **Research Approach**: Design Science Research framework with iterative development
- **Requirements**: Educational and functional requirements derived from learning objectives
- **Technology Selection**: Justified choices of Godot Engine, GDScript, GDSync, and HTTP/SSE relay architecture based on project needs
- **Game Design**: Systematic mapping of HPC concepts to game mechanics with UX design principles
- **Evaluation**: Multi-faceted evaluation approach combining technical, usability, and educational assessment

The next chapter presents the detailed system architecture resulting from these methodological decisions, including scene structure, component design, and multi-player networking architecture.

Chapter 4

System Design and Architecture

This chapter presents the system architecture decisions that enabled the HPC Sorting Serious Game to be feasible, maintainable, and pedagogically interpretable. It focuses on architectural structure and rationale rather than exhaustive implementation listings. Detailed implementation is provided in Chapter 5, and educational/operational effectiveness is evaluated in Chapter 6.

4.1 Architectural Overview

The architecture is organized to support two complementary goals:

- **Pedagogical clarity**: game mechanics must remain interpretable as OpenMP-inspired shared-memory collaboration.
- **Technical robustness**: multiplayer synchronization must remain consistent across browser clients under variable latency.

4.1.1 Design Principles

The architecture is guided by the following principles:

1. **Component modularity**: cards, buffers, and managers are separated so mechanics can evolve without rewriting the whole system.
2. **Separation of concerns**: UI, game rules, and synchronization logic are isolated to reduce coupling during iterative experiments.
3. **Host-authoritative conflict handling**: one authority resolves contested actions to prevent divergent multiplayer states.
4. **Observable synchronization**: architecture includes explicit state transitions that can be logged and inspected during debugging.

5. **Web-first constraints**: transport, update cadence, and payload design prioritize browser compatibility over engine-native networking assumptions.

4.1.2 System Layers

The system is organized into four layers, each with a distinct responsibility:

- **Presentation Layer**: Renders cards, buffers, indicators, and interaction feedback.
- **Game Logic Layer**: Validates sorting actions, completion conditions, and per-mode rules.
- **Synchronization Layer**: Coordinates shared/private state propagation, ordering, and conflict resolution.
- **Transport/Relay Layer**: Handles browser-compatible signaling and event relay via HTTP/SSE

These layers decouple pedagogical behavior from transport mechanics, which was essential when adapting multiplayer behavior for browser export.

4.2 Scene and Interaction Architecture

The scene model is intentionally concise: a menu/lobby entry path and two gameplay contexts (single-player baseline and multiplayer OpenMP-style collaboration). The scene split supports direct comparison between sequential and collaborative execution while reusing core card and validation logic.

Architectural Intent:

- Single-player provides the sequential baseline for speedup interpretation.
- Multiplayer preserves shared container semantics while enforcing local buffer visibility.
- Lobby separates session orchestration from gameplay state transitions.

This scene structure keeps learning signals (who can see what, who can move what, when state is final) explicit for both users and evaluators.

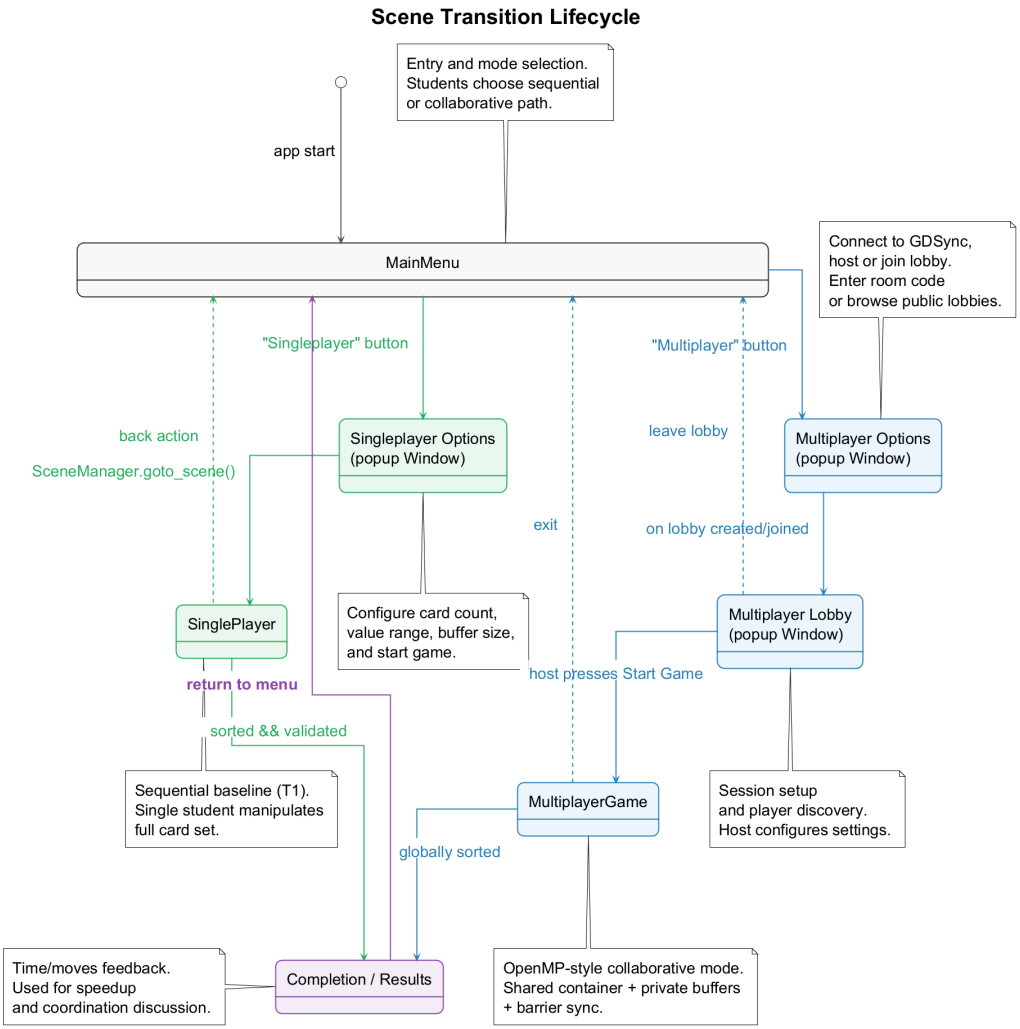


Figure 4.1: Scene lifecycle transitions used by the game flow, including single-player baseline and multiplayer lobby path

4.2.1 Scene Lifecycle and Transition Logic

Scene changes are managed through a central scene manager to avoid ad-hoc transitions across gameplay scripts. This keeps navigation behavior deterministic and makes it easier to relate transitions to teaching intent (e.g., sequential baseline first, then collaborative mode).

4.2.2 Event-Driven Coordination

The architecture is event-driven at component boundaries. Local game objects communicate via Godot signals, while multiplayer state changes are propagated through authority-managed synchronization paths. This split allows UI responsiveness without directly coupling UI events to transport internals.

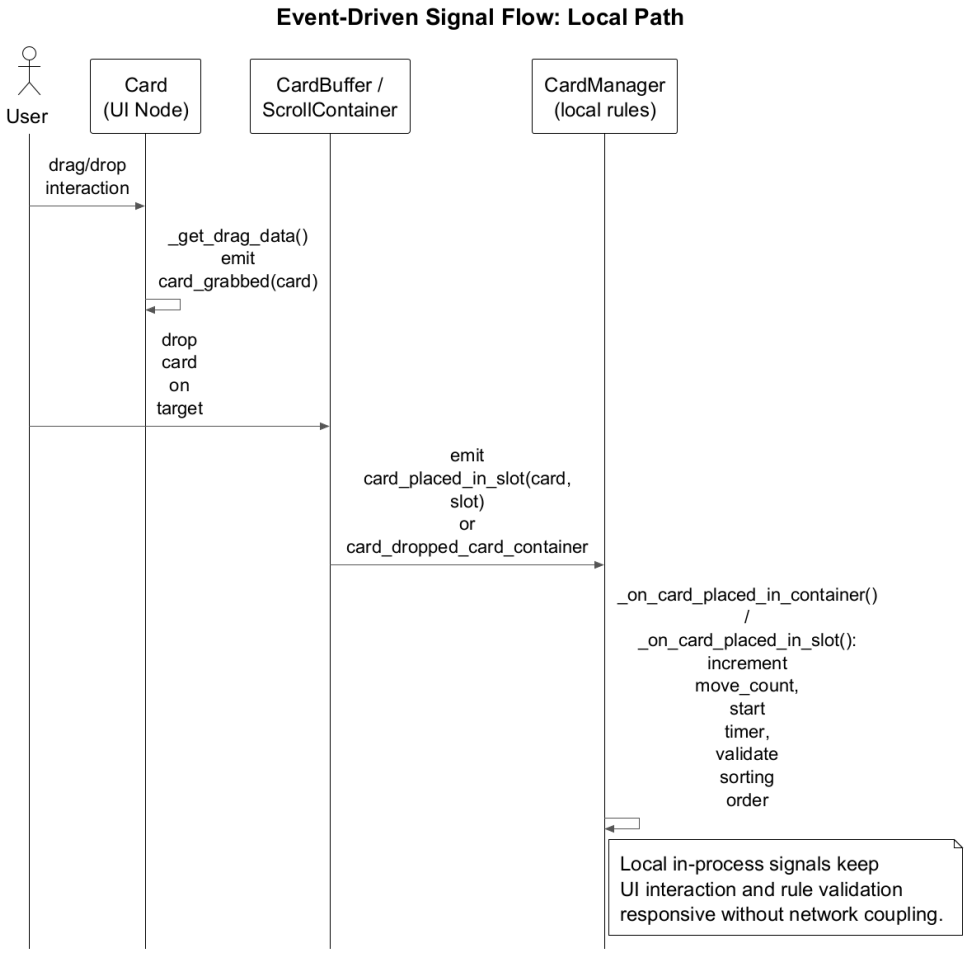


Figure 4.2: Event-driven local signal path across UI interaction and rule validation components

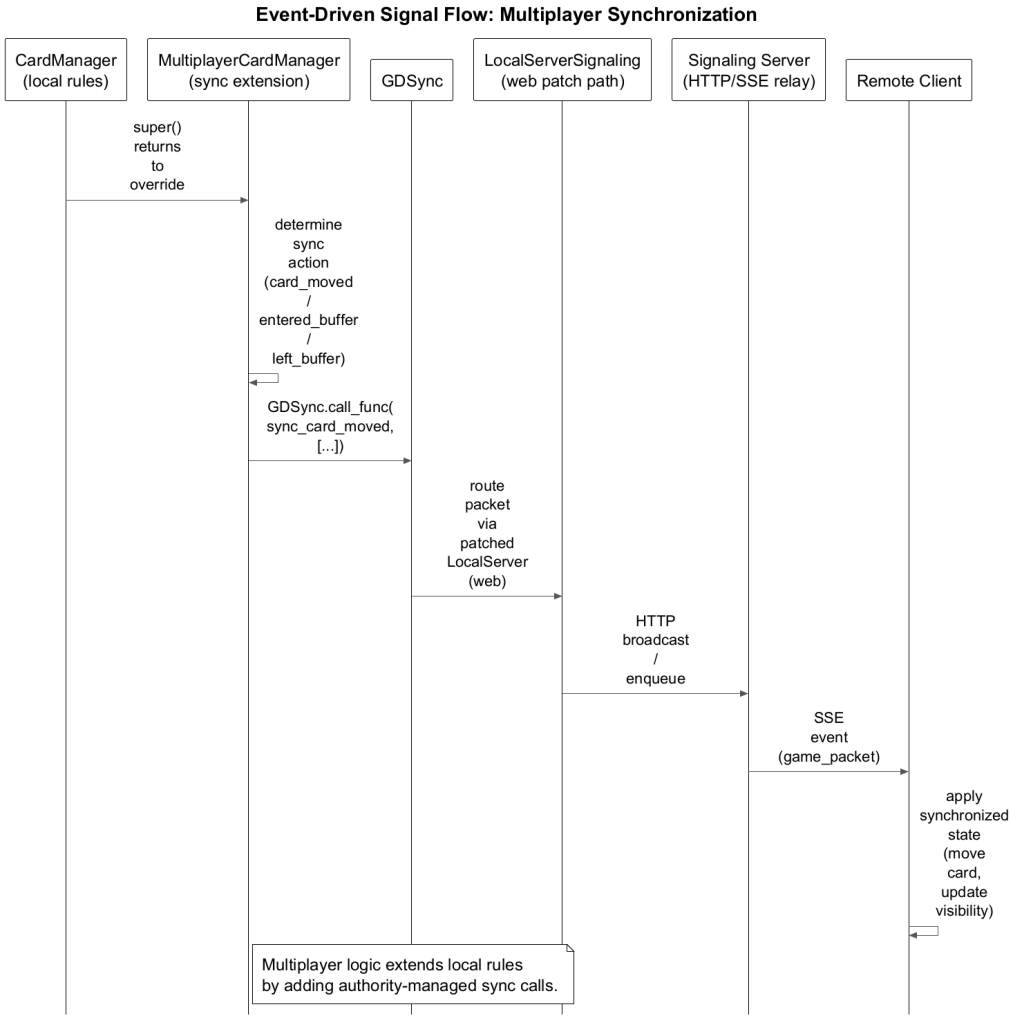


Figure 4.3: Multiplayer synchronization path extending local rule handling through relay-based transport

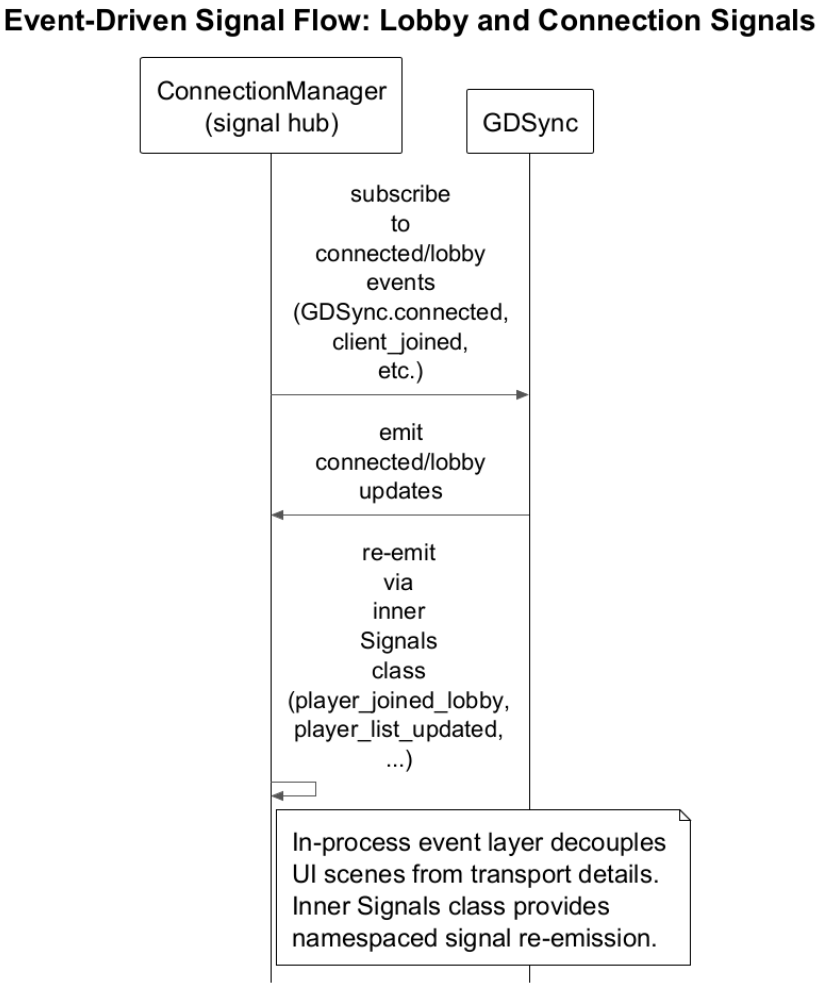


Figure 4.4: Lobby and connection signal path with namespaced re-emission in the connection manager

4.3 Core Component Roles

Rather than duplicating full class implementations, this section summarizes architecture-critical component responsibilities.

Card Component (Interaction Unit):

- Encapsulates value identity, interaction state, and visual feedback states.
- Emits events consumed by manager-level logic rather than mutating global state directly.

Card Manager (Rule and State Coordinator):

- Owns sorting validity checks and authoritative placement updates.
- Serves as the integration boundary between UI interaction and synchronization rules.
- Was designed for reusability: shared logic lives in `card_manager.gd` so mode-specific behavior can be layered instead of duplicated.

Multiplayer Card Manager (Authority Extension):

- Explicitly extends `card_manager.gd` as `multiplayer_card_manager.gd`, adding host-authoritative synchronization and reconciliation behavior while reusing baseline card/rule logic.
- Enforces shared/private visibility semantics across peers.

4.4 Multiplayer Architecture

4.4.1 Network Topology

The implemented multiplayer topology uses browser-compatible relay patterns for session events and state updates. This avoids assumptions that are unstable in browser-exported runtime contexts.

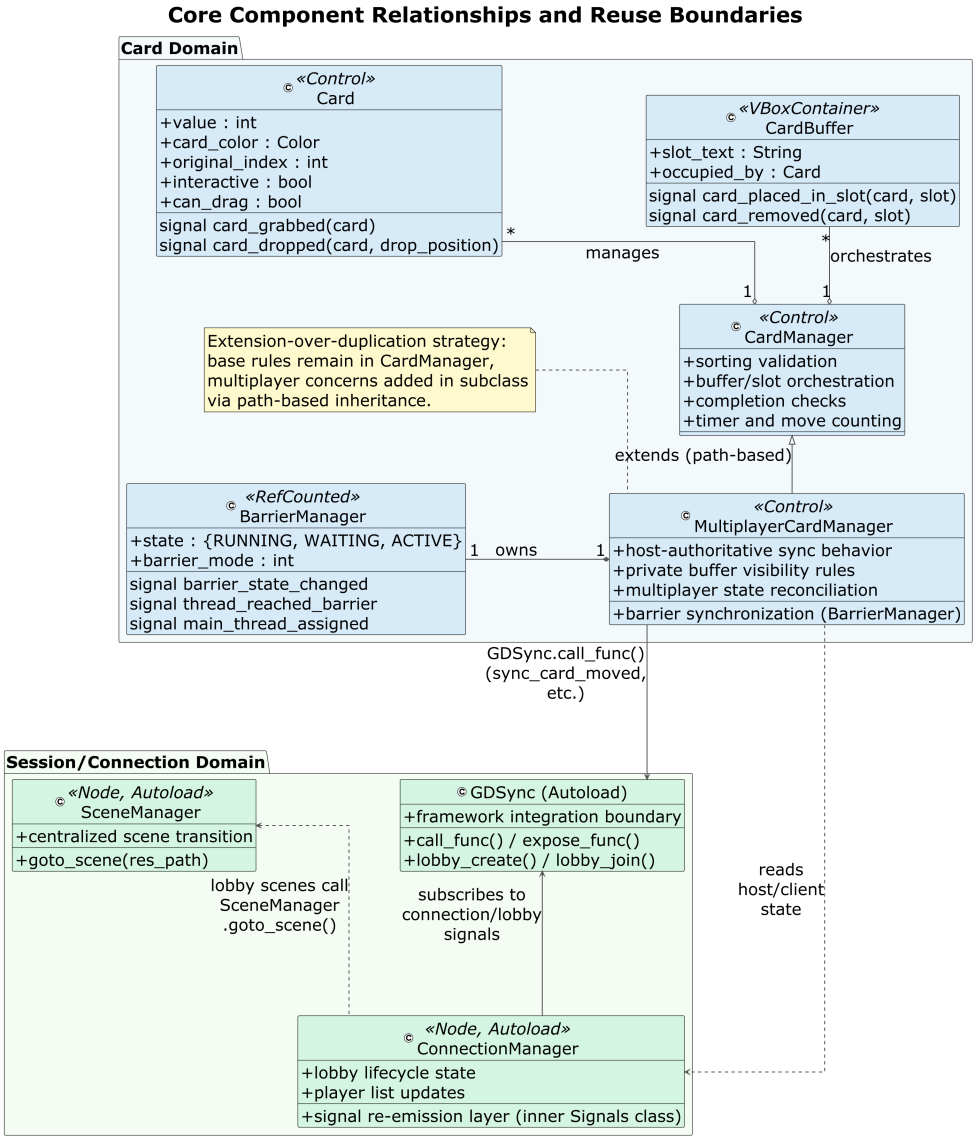


Figure 4.5: Component and inheritance overview showing reuse boundary between `CardManager` and `MultiplayerCardManager`

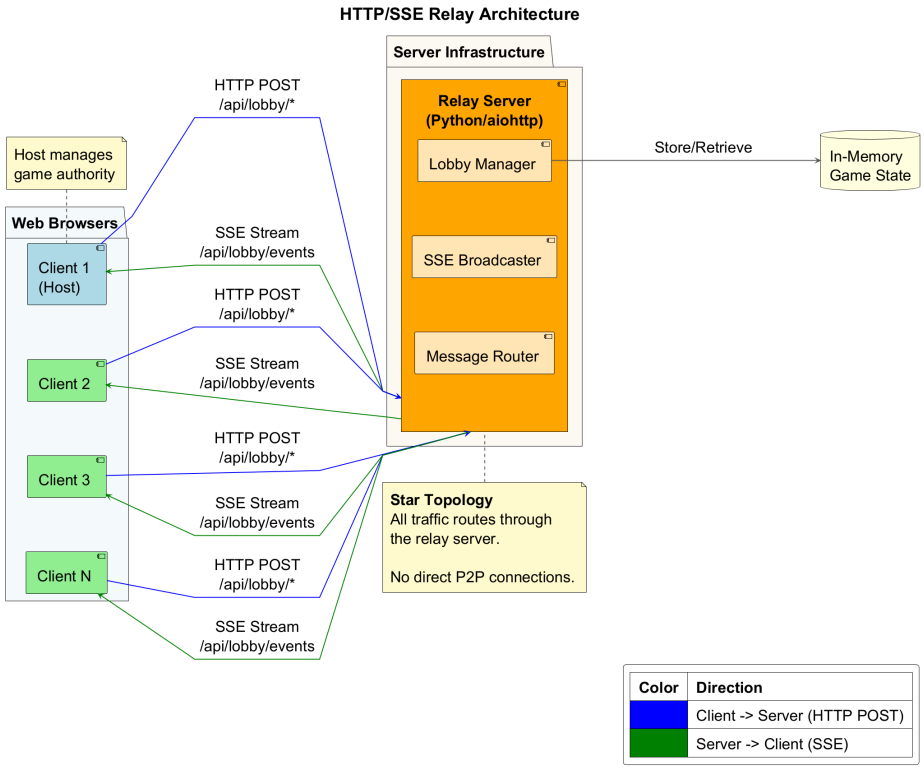


Figure 4.6: HTTP/SSE relay-oriented architecture used for web-first multiplayer synchronization

4.4.2 State Synchronization Patterns

Two synchronization patterns are central:

- **Optimistic local interaction**: local responsiveness is preserved while awaiting authoritative confirmation.
- **Host broadcast reconciliation**: final shared state is re-broadcast to all peers to converge state.

```
# Client proposes move -> host validates -> host broadcasts
    canonical state
on_client_move(card_id, target_slot):
    if host_validate(card_id, target_slot):
        host_apply(card_id, target_slot)
        broadcast_canonical_state(card_id, target_slot)
    else:
        reject_and_restore(card_id)
```

Listing 4.1: Architectural pattern: host-authoritative move confirmation (pseudocode)

This pattern minimizes permanent divergence while preserving interaction fluidity.

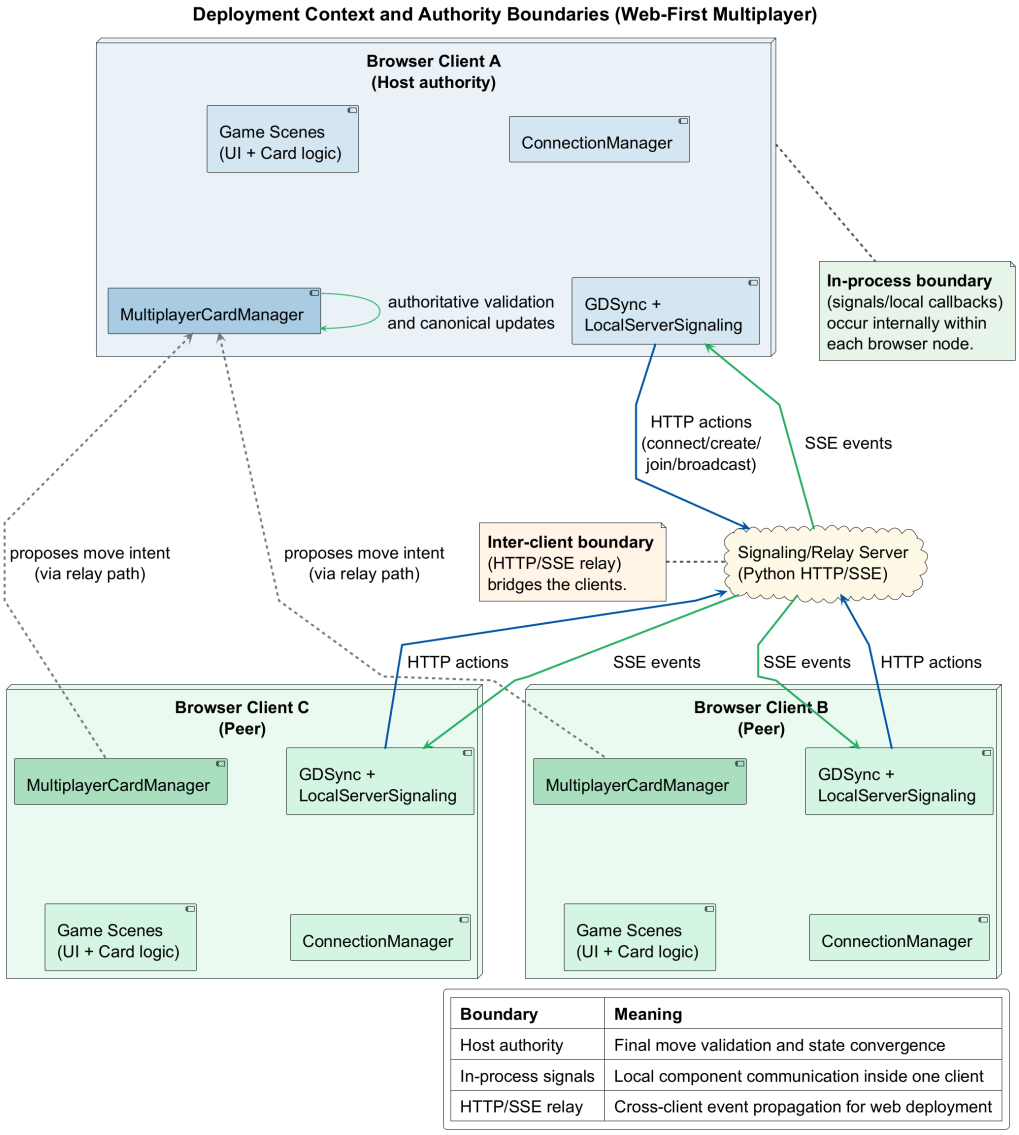


Figure 4.7: Runtime deployment context with host authority and shared/private synchronization boundaries

4.4.3 GDSync Integration Boundary

GDSync is treated as a synchronization abstraction layer, while architecture ownership remains in the game managers. This prevented framework-specific behavior from leaking into all gameplay components and reduced replacement cost when debugging framework limitations. The web-export signaling patch details are intentionally analyzed in Chapter 7, Section 7.3.3, while this section keeps the architectural boundary view.

```
@GDSync.rpc(call_local=true)
func sync_move(card_id: int, target_id: int):
    multiplayer_manager.apply_authoritative_move(card_id, target_id
    )
```

Listing 4.2: Architectural boundary: sync wrapper delegates to game manager (pseudocode)

4.4.4 Connection Management

Connection handling is split into phases:

- **Lobby phase**: peer discovery, readiness, and session configuration.
- **Gameplay phase**: synchronized interaction and reconciliation.
- **Failure phase**: disconnection detection and graceful degradation.

4.5 Data Flow and Visibility Model

The architecture explicitly distinguishes:

- **Shared-visible state**: container cards and game-level counters.
- **Player-private state**: cards in local buffers hidden from other players.

This model reflects OpenMP's shared-memory paradigm: all threads operate on a common address space, while thread-local buffers remain private to each thread.

```
if card.in_private_buffer():
    card.visible = (card.buffer_owner_id == local_player_id)
else:
    card.visible = true
```

Listing 4.3: Architectural mechanism: local buffer visibility gate (pseudocode)

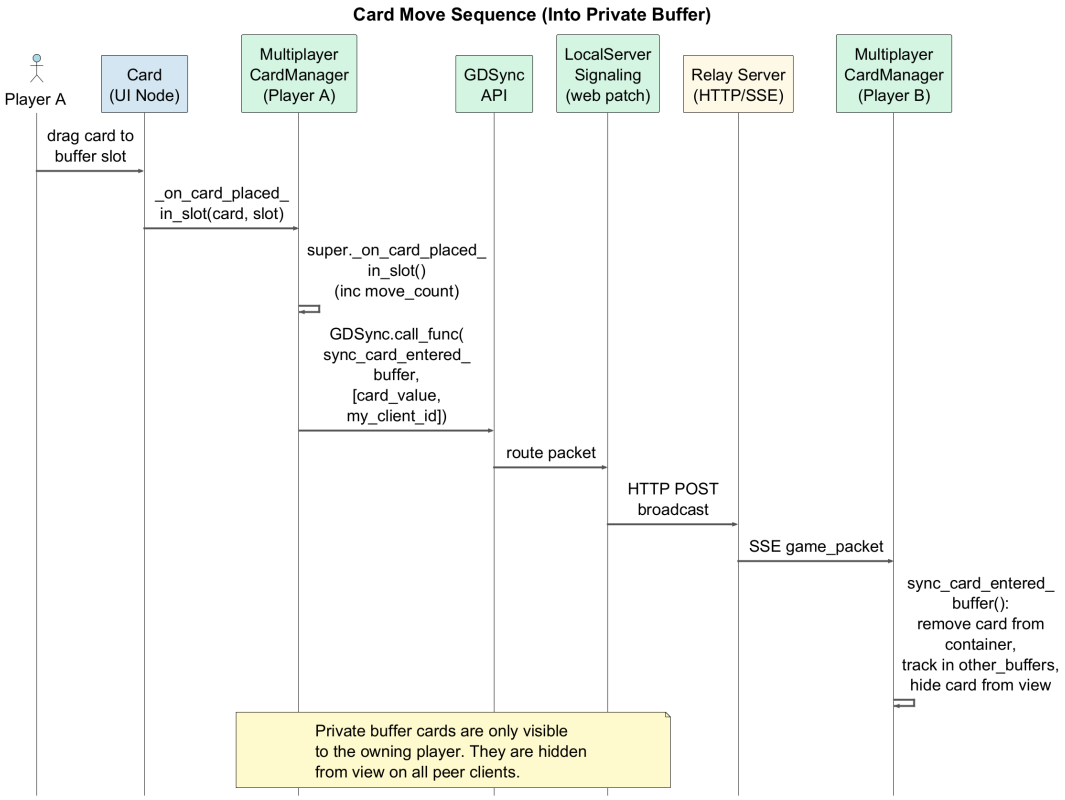


Figure 4.8: Data flow when a card enters a player buffer (proposal phase)

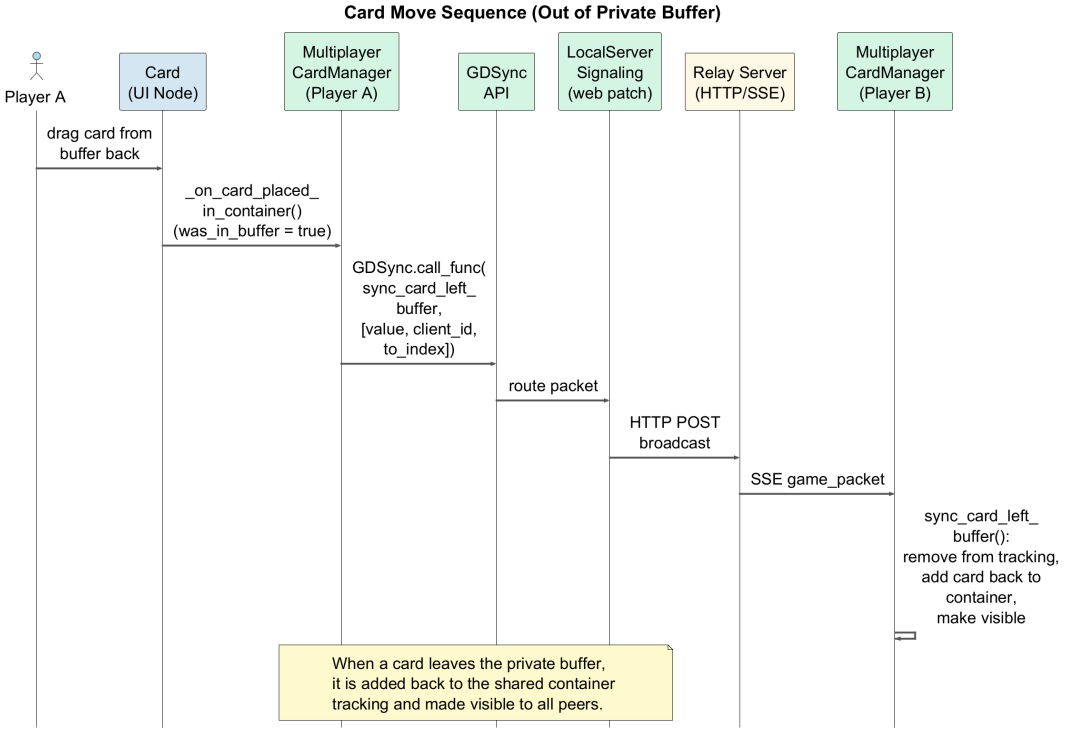


Figure 4.9: Data flow when a card leaves a player buffer (confirmation phase)

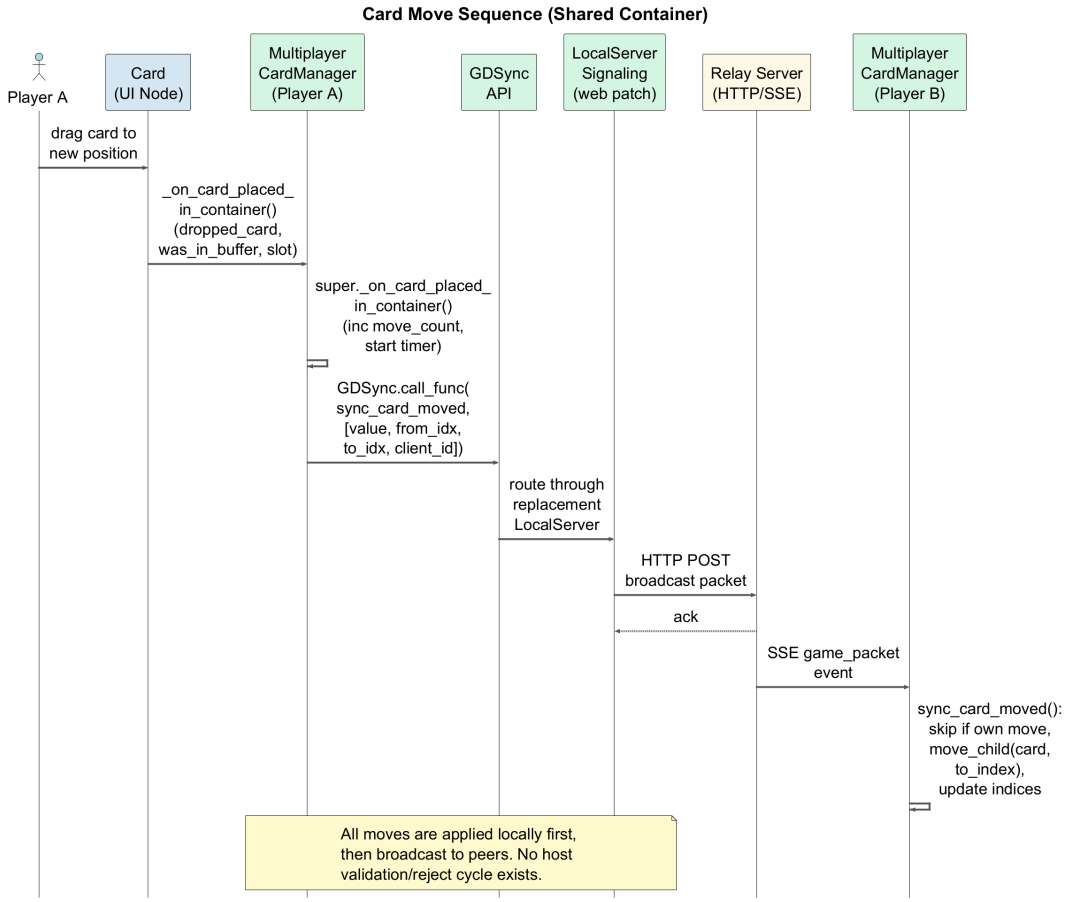


Figure 4.10: Data flow in the shared container during authoritative update and visibility adjustment

4.6 Barrier Synchronization Architecture

The game includes a barrier synchronization feature that directly models the `#pragma omp barrier` construct from OpenMP. Architecturally, the barrier is implemented as a separate `BarrierManager` component (not embedded in the card manager) following the single-responsibility principle.

4.6.1 State Machine Design

The barrier operates as a three-state finite automaton with the following transitions:

1. **RUNNING → WAITING_AT_BARRIER**: Any player initiates a barrier; all must arrive.
2. **WAITING_AT_BARRIER → BARRIER_ACTIVE**: All players have reached the barrier; the main thread is granted exclusive access to the shared container.
3. **BARRIER_ACTIVE → RUNNING**: The main thread releases the barrier; normal play resumes.

This maps directly to OpenMP barrier semantics: threads reaching the barrier must wait until all threads have arrived, after which execution proceeds in a coordinated fashion.

4.6.2 Integration Points

The `BarrierManager` integrates with the multiplayer card manager through three mechanisms:

- **Signals**: emits `barrier_state_changed` to trigger UI overlays (lock screen for non-main-thread players).
- **GDSync exposure**: barrier functions are registered for remote invocation alongside card synchronization functions.
- **Settings**: BarrierManager reads the barrier mode (first-to-reach or round-robin) from the global `Settings` autoload.

The barrier implementation details are covered in Chapter 5, Section 5.5.

4.7 Responsive UI/UX Architecture

UI architecture prioritizes interpretable interaction under constrained viewports:

- Scrollable container strategy for large card sets.
- Buffer-first layout to reinforce shared/private distinction.
- Immediate feedback (toast/status indicators) to reduce action ambiguity in multiplayer.

4.7.1 UI Layering and Overlay Isolation

Overlay and inspection elements are separated from gameplay nodes to minimize accidental coupling between debug/auxiliary UI and gameplay transforms. This layering approach supports live inspection tooling while preserving gameplay interaction consistency.

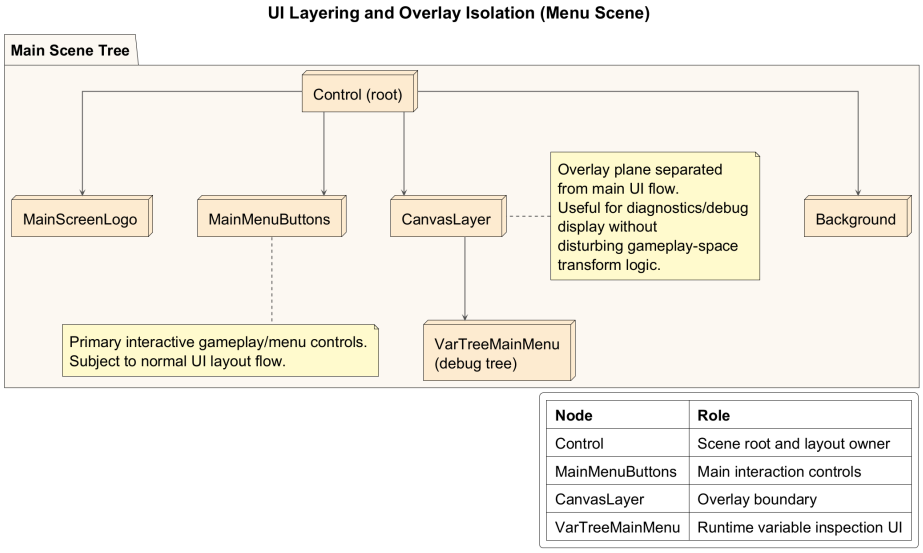


Figure 4.11: UI layering with CanvasLayer-based overlays separated from gameplay-space nodes

A playful evasive hover animation is also present in the main menu as a non-critical engagement microinteraction; its reception in early tests is discussed in Chapter 6, Section 6.1.5.

4.8 Performance and Scalability Constraints

Architecture-level performance strategies include:

- bounded synchronization payloads,
- authoritative reconciliation instead of full scene replication,
- UI/container policies that remain usable at higher card counts.

Observed constraints and measured outcomes are discussed in Chapter 6.

4.9 Architectural Support for Parallel Sorting Strategies

The game does not enforce a specific sorting algorithm. Instead, its architecture provides general-purpose mechanics—a shared card container, player local buffers, and barrier synchronization—that support **static partitioning strategies** such as parallel merge sort. As established in the domain model (Section 3.2.5), the game does *not* support recursive strategies such as parallel quicksort, which require dynamic pivot selection and multi-round data redistribution. This section demonstrates how parallel merge sort can be carried out within the existing tool without code changes, explains why quicksort falls outside the current mechanics, and identifies what architectural extensions would be needed to support it in the future.

4.9.1 How the Existing Mechanics Support Static Partitioning

Three existing architectural components form the substrate for static partitioning strategies:

- **Shared card container (shared memory)**: All cards are visible and draggable in a common horizontal container. Any player can reorder any card at any time, just as any thread can access shared memory. This allows collaborative rearrangement and independent sub-array sorting.
- **Player local buffers (partitioned shared memory)**: Each player has local buffer slots for temporarily holding cards. The buffer contiguity validation enforces that a player works on a logically coherent subset of the data. Players use buffers for local sorting before returning cards to the shared container.
- **Barrier synchronization (`#pragma omp barrier`)**: The `BarrierManager`'s three-state FSM (`RUNNING` → `WAITING_AT_BARRIER` → `BARRIER_ACTIVE`) lets players coordinate between algorithm phases—e.g., after local sorting before merging. The main-thread selection mechanism assigns the coordinator role to the first player to reach the barrier.

These components directly support the "divide into intervals, sort locally, assemble" workflow described in the domain model (Section 3.2). They do *not* support recursive repartitioning, dynamic pivot negotiation, or task-based sub-group splitting required by quicksort.

4.9.2 Performing Parallel Merge Sort with the Tool

Players can execute a parallel merge sort strategy using the existing mechanics (Figure 4.13):

1. **Divide**: Players informally agree to each "own" a contiguous segment of the card deck (e.g., with 4 players and 20 cards, each player takes cards 1–5, 6–10, etc.).

2. **Local sort**: Each player moves their assigned cards into their local buffer, sorts them locally, then returns them to the shared container in sorted order. All players do this simultaneously—embarrassingly parallel local with no inter-player communication.

3. **Merge**: Players use the barrier to synchronize. One player (the "main thread") merges two adjacent sorted runs while the other player watches. Subsequent barrier rounds merge progressively larger runs until the full array is sorted.

The pedagogical mappings that emerge naturally from this strategy:

- *Data decomposition*: players divide cards among themselves ≡ `#pragma omp parallel for` static scheduling
- *Embarrassingly parallel local sort*: independent work in local buffers ≡ no inter-thread communication
- *Barrier-gated merge rounds*: using the barrier between merge phases ≡ `#pragma omp barrier`
- *Idle threads*: non-merger players blocked at barrier ≡ synchronization overhead

4.9.3 Why Parallel Quicksort Is Not Supported

Parallel quicksort requires capabilities that fall outside the current game mechanics:

1. **Dynamic pivot selection**: Players would need a mechanism to collectively agree on a pivot value at each recursion level. The current game has no in-game communication channel for negotiating pivots—players can only move cards.

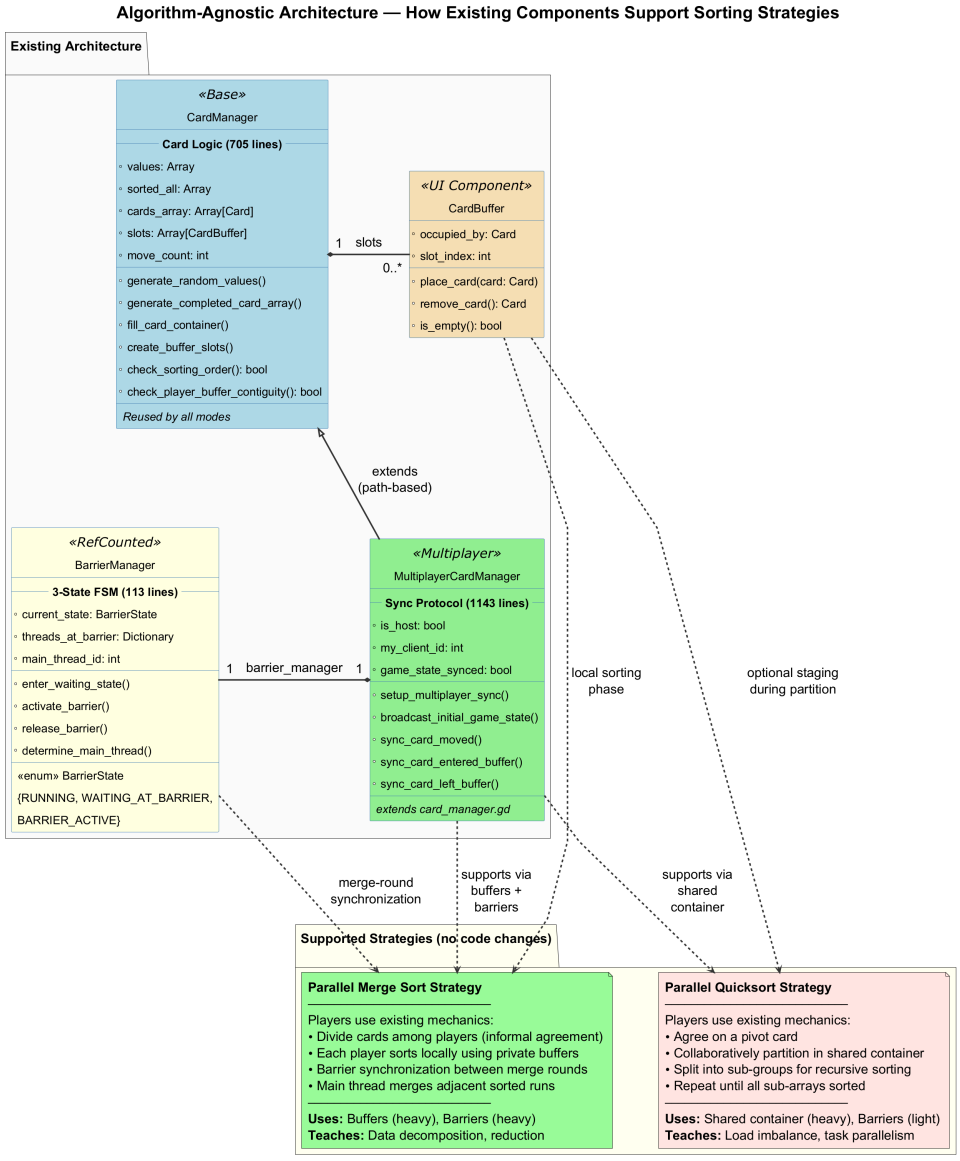


Figure 4.12: Existing component architecture and how it supports parallel sorting strategies: `CardManager` provides card logic, `MultiplayerCardManager` adds synchronization, `BarrierManager` provides coordination, and `CardBuffer` provides per-thread work areas—together supporting static partitioning strategies such as parallel merge sort

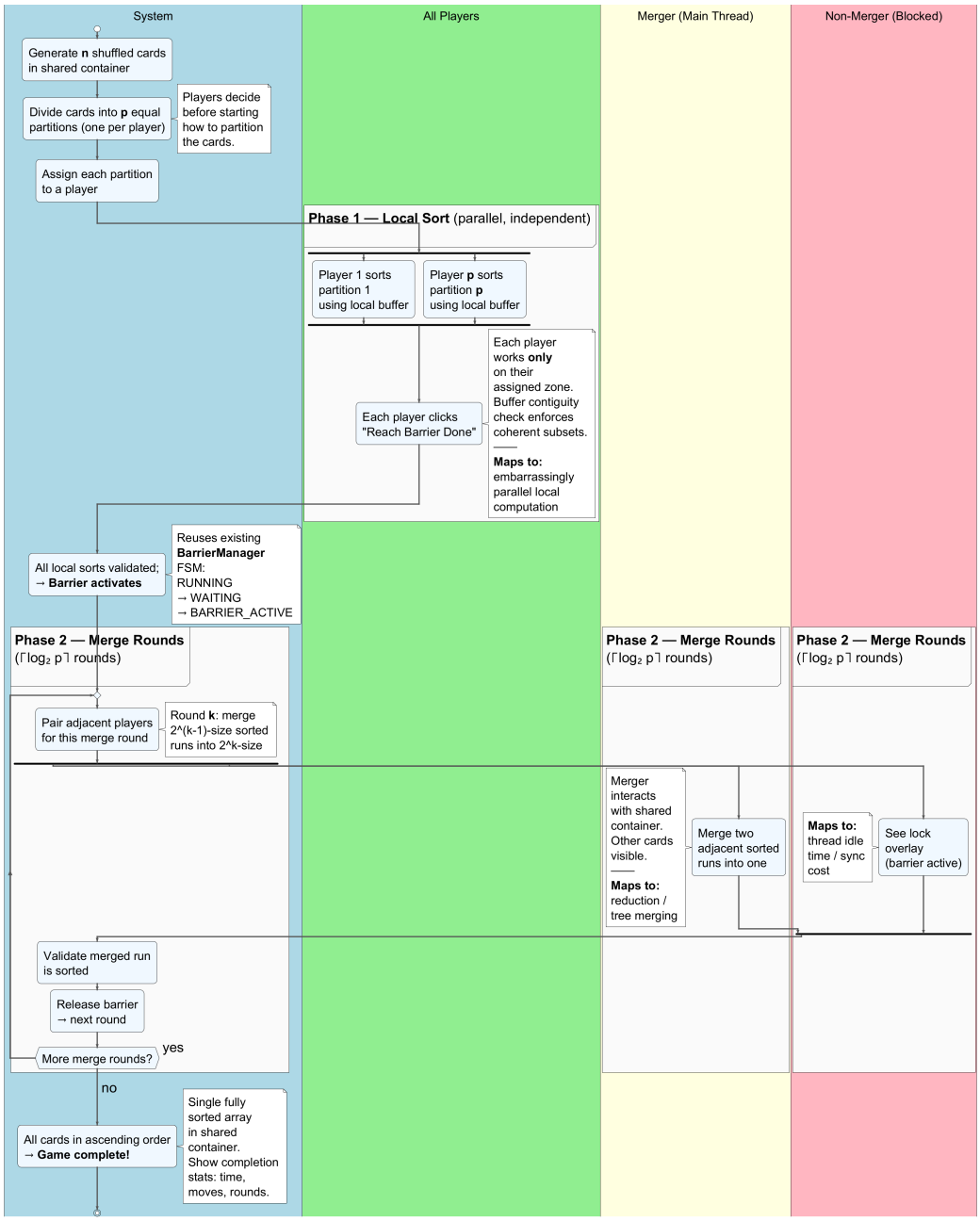


Figure 4.13: How players perform parallel merge sort using the existing tool: divide cards among players, sort locally in local buffers, then merge sorted runs through barrier-gated rounds

2. **Multi-round recursive repartitioning**: After each pivot-based partition, the remaining sub-arrays must be recursively subdivided among smaller sub-groups. The game's single-round "pick your interval, sort locally, assemble" workflow has no concept of recursive sub-rounds or dynamic sub-group formation.

3. **Task-based sub-group splitting**: In parallel quicksort, the set of threads working on a sub-problem changes at each recursion level. The current barrier mechanism synchronizes *all* players simultaneously; it cannot synchronize arbitrary subsets independently.

Figure 4.14 illustrates how players *might conceptually* perform quicksort if the mechanics were extended, but the current tool does not enforce or support this workflow.

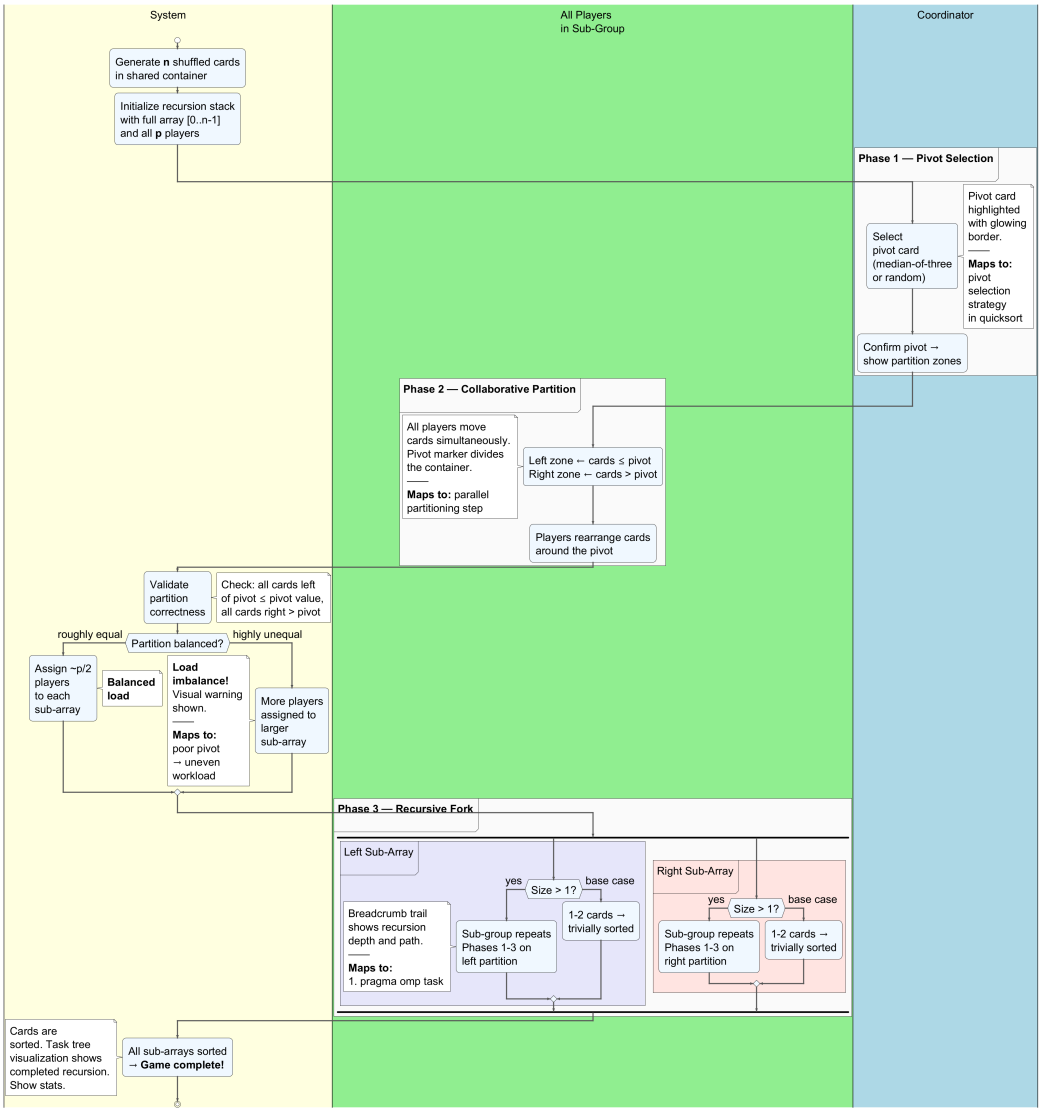


Figure 4.14: Conceptual illustration of how parallel quicksort *could* be performed if the game were extended with dynamic pivot selection, recursive sub-rounds, and sub-group barriers. This workflow is **not currently supported** by the tool.

Extending the game to support guided quicksort is identified as future work in Chapter 8, Section 8.4.2.

4.9.4 Comparison: Merge Sort (Supported) vs. Quicksort (Not Supported)

Table 4.1 summarizes why parallel merge sort is supported by the existing mechanics while parallel quicksort is not.

Table 4.1: Comparison of parallel sorting strategies: merge sort is supported; quicksort requires extensions

Aspect	**Merge Sort (Supported)**	**Quicksort (Not Supported)**
Parallelism model	Data parallelism (equal partitions)	Task parallelism (recursive fork)
Partition method	Static, pre-agreed intervals	Dynamic, pivot-dependent
Barrier usage	Between merge rounds (supported)	Between partition rounds (not supported—requires sub-group barriers)
Buffer usage	Heavy—local sorting in local buffers	Light—partition done in shared container
Load balance	Always balanced (equal partitions)	Variable—depends on pivot quality
Primary HPC lesson	Barrier sync, reduction pattern	Load imbalance, task creation
Game support	**Yes**—existing mechanics sufficient	**No**—requires pivot UI, recursive rounds, sub-group barriers

4.10 Summary

This chapter documented architectural decisions that made the tool technically feasible and pedagogically interpretable in a web-first multiplayer context. The focus was on synchronization boundaries, state visibility semantics, and authority patterns rather than exhaustive implementation detail. Section 4.9 demonstrated that the existing architecture—the shared card container, player work buffers, and barrier synchronization—supports **static partitioning strategies** such as parallel merge sort. Parallel quicksort is *not* supported by the current mechanics due to its requirement for dynamic pivot selection, recursive repartitioning, and sub-group

barriers; extending the tool to support quicksort is discussed as future work in Chapter 8. Detailed implementation evidence is provided in Chapter 5, while educational and operational effectiveness is evaluated in Chapter 6.

Chapter 5

Implementation

This chapter presents key implementation mechanisms of the HPC Sorting Serious Game. It covers the core game logic, multiplayer synchronization protocol, barrier synchronization (a direct pedagogical mapping to `#pragma omp barrier`), the web-export compatibility patch, and development observability tooling. Tool/framework selection rationale is documented in Chapter 3, architectural decisions in Chapter 4, and effectiveness evaluation in Chapter 6.

The complete source code is available in the project repository [46].

5.1 Implementation Focus Areas

The implementation evidence is organized around six high-impact concerns:

- card management and sorting correctness,
- component reuse through inheritance (`CardManager` → `MultiplayerCardManager`),
- network state serialization and synchronization protocol,
- barrier synchronization as a pedagogical feature,
- web-export compatibility patching,
- observability and debuggability for multiplayer faults.

5.2 Card Management and Sorting Correctness

The single-player card management logic resides in `card_manager.gd` (705 lines), which serves as both the standalone game controller and the base class for multiplayer extension.

5.2.1 Card Generation and Initialization

Card generation supports configurable counts (1–200) with optional unique or repeated values. The initialization sequence in `_ready()` follows a deterministic pipeline:

```
func _ready():
    if num_cards < 1:
        num_cards = 1
    values = generate_random_values()
    sorted_all = values.duplicate() # Create a separate sorted
    array for reference
    sorted_all.sort()

    adjust_container_spacing()
    cards_array = generate_completed_card_array(values)
    sorted_cards_array = generate_completed_card_array(sorted_all,
    "SortedCard_")
    fill_card_container(cards_array, card_container)
    fill_card_container(sorted_cards_array, sorted_cards_container)
    slots = create_buffer_slots()
    _connect_signals()
```

Listing 5.1: Card manager initialization pipeline (card_manager.gd)

Each card is instantiated from a preloaded scene, assigned a numeric value and a color derived from the value (for visual distinctiveness), and placed in a scrollable `HBoxContainer`. A parallel sorted reference array is generated so players can optionally view the target ordering.

5.2.2 Sorting Validation

Sorting validation is the completion contract used in both single-player and multiplayer modes. The actual implementation iterates over the card container's children (the authoritative UI ordering):

```
func check_sorting_order() -> bool:
    var sorted_correctly = true
    if card_container.get_child_count() != num_cards:
        return false
    var cards_in_container: Array[Card] = []
    for child in card_container.get_children():
        if child is Card:
            cards_in_container.append(child as Card)
    for i in range(1, cards_in_container.size()):
        var current_card = cards_in_container[i].value
        var previous_card = cards_in_container[i - 1].value
        if current_card < previous_card:
            sorted_correctly = false
            break
    return sorted_correctly
```

Listing 5.2: Sorting validation (card_manager.gd, line 645)

This mechanism provides a deterministic completion criterion independent of UI animation state and network timing. Notably, validation checks the *scene tree child order* rather than an abstract array, ensuring that what the player sees matches what is validated (see Chapter 6, Section 6.5).

5.2.3 Buffer Contiguity Check

Players use local buffer zones (simulating partitioned shared memory) to temporarily hold cards during sorting. A buffer contiguity check validates that cards in the buffer form a contiguous subsequence of the sorted target:

```
func check_player_buffer_contiguity() -> bool:
    var buffer_values = []
    for slot in slots:
        if slot.occupied_by == null:
            return false
        buffer_values.append(slot.occupied_by.value)
    return SubarrayUtils.is_contiguous_subarray(sorted_all,
   buffer_values)
```

Listing 5.3: Buffer contiguity validation (card_manager.gd, line 693)

This constraint reinforces the pedagogical mapping: just as a thread in an OpenMP program should work on a logically coherent subset of data, players must select contiguous card ranges for their buffers.

5.3 Component Reuse: CardManager Inheritance

The multiplayer game controller, `MultiplayerCardManager` (1143 lines), explicitly extends the single-player `CardManager`:

```
extends "res://scenes/CardScene/scripts/card_manager.gd"
class_name MultiplayerCardManager
```

Listing 5.4: Multiplayer extension declaration (multiplayer_card_manager.gd, line 1)

This inheritance structure allows the multiplayer mode to reuse all card generation, UI layout, sorting validation, and buffer management logic while adding:

- host/client role differentiation,
- network state serialization (via `CardState`),
- GDSync function exposure for remote procedure calls,
- barrier synchronization integration.

The multiplayer `_ready()` overrides the parent to branch on host vs. client role:

```
func _ready():
    is_host = ConnectionManager.am_i_host()
    my_client_id = ConnectionManager.get_my_client_id()

    barrier_manager = BarrierManager.new()
    barrier_manager.set_barrier_mode(Settings.barrier_mode)
    barrier_manager.barrier_state_changed.connect(
    _on_barrier_state_changed)

    setup_multiplayer_sync()

    if is_host:
        super._ready()
        await get_tree().process_frame
        broadcast_initial_game_state()
        game_state_synced = true
    else:
        _initialize_client_structure()
        await get_tree().create_timer(0.5).timeout
        request_game_state_from_host()
```

Listing 5.5: Multiplayer initialization with host/client (multiplayer_card_manager.gd)

The host executes the parent `_ready()` (generating cards normally) and then broadcasts the initial state. Clients initialize an empty UI structure and request the authoritative game state from the host, avoiding duplicate card generation that would cause divergence.

5.4 Multiplayer Synchronization Protocol

5.4.1 CardState Serialization

All card state transmitted over the network is serialized through a dedicated `CardState` inner class:

```
class CardState:
    var value: int
    var index: int
    var original_index: int
    var in_container: bool
    var in_buffer: bool
    var buffer_owner: int

    func to_dict() -> Dictionary:
        return {
            "value": value,
            "index": index,
            "original_index": original_index,
```

```
14              "in_container": in_container,
15              "in_buffer": in_buffer,
16              "buffer_owner": buffer_owner
17          }
18
19      static func from_dict(data: Dictionary) -> CardState:
20          return CardState.new(
21              data.get("value", 0),
22              data.get("index", -1),
23              data.get("original_index", -1),
24              data.get("in_container", true),
25              data.get("in_buffer", false),
26              data.get("buffer_owner", -1)
27          )
```

Listing 5.6: CardState transport protocol (multiplayer_card_manager.gd, lines 6–49)

`CardState` captures whether a card is in the shared container or in a player's local buffer, and who owns it. This explicit serialization boundary ensures that network payloads are minimal and self-describing, and that deserialization on the receiving client produces a deterministic scene state.

5.4.2 Lobby and Session Lifecycle

The lobby manages session creation/joining and readiness transitions before gameplay authority is established.

1. Player creates room or enters room code
2. Connection to signaling server via HTTP/SSE
3. Peer handshake with host
4. Players appear in lobby list
5. Host configures game settings (card count, buffer size, barrier mode)
6. Host starts game, triggering scene transition to multiplayer gameplay

5.4.3 GDSync Function Exposure

GDSync provides high-level networking abstractions. For this thesis, the project uses an author-maintained GDSync fork that includes a fix required for the web-export workflow [34]. A pull request with this fix was submitted upstream [32].

Rather than using decorator-based annotations, the actual GDSync integration uses explicit function exposure. The `setup_multiplayer_sync()` method registers all remotely callable functions:

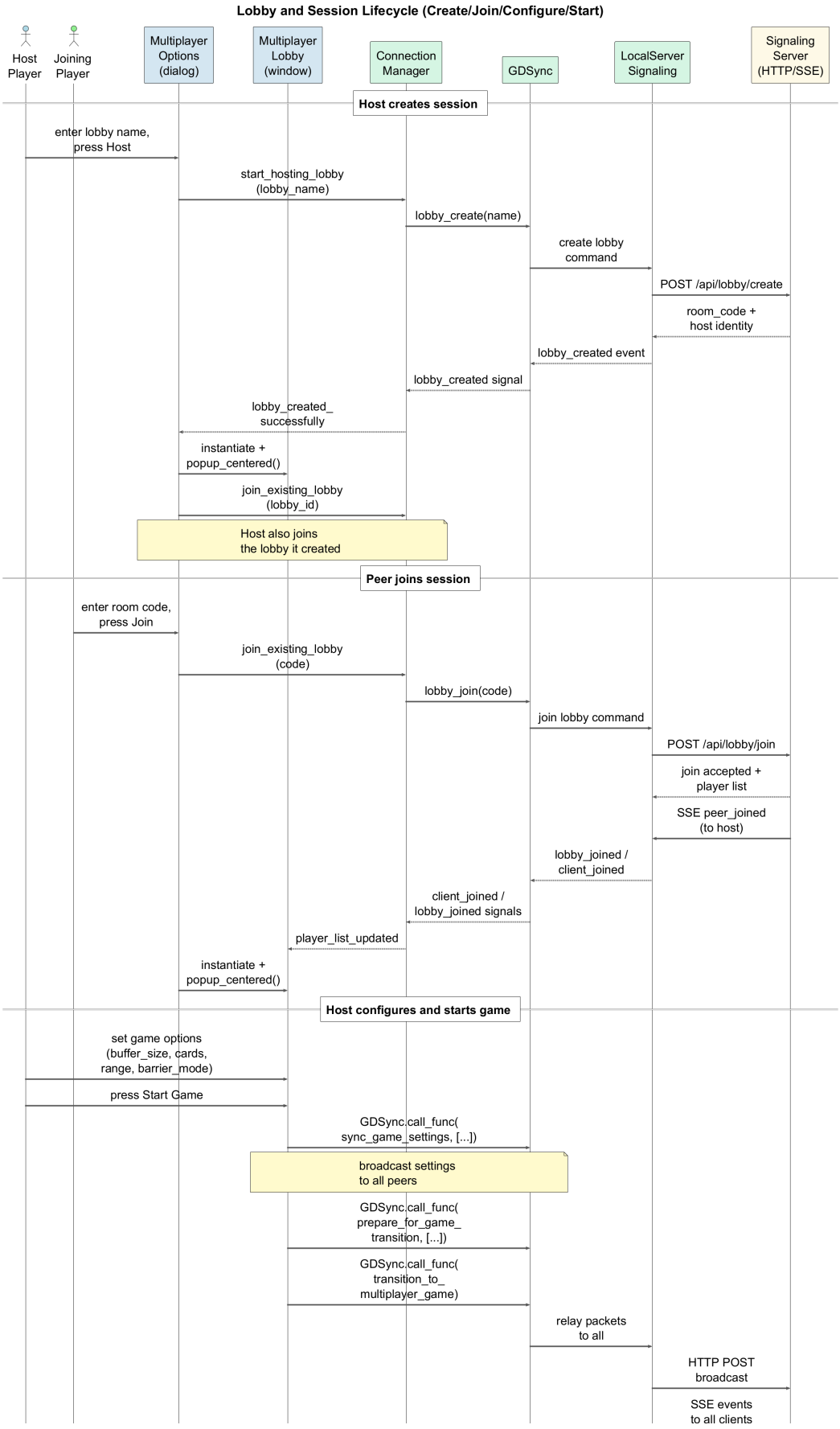


Figure 5.1: Lobby and session lifecycle from room creation/join to synchronized game start

```gdscript
func setup_multiplayer_sync():
    GDSync.expose_func(self.sync_complete_game_state)
    GDSync.expose_func(self.sync_card_moved)
    GDSync.expose_func(self.sync_card_entered_buffer)
    GDSync.expose_func(self.sync_card_left_buffer)
    GDSync.expose_func(self.sync_timer_state)
    GDSync.expose_func(self.sync_game_finished)
    GDSync.expose_func(self.send_current_state_to)
    # Barrier synchronization functions
    GDSync.expose_func(self.barrier_thread_reached)
    GDSync.expose_func(self.barrier_activate)
    GDSync.expose_func(self.barrier_card_picked)
    GDSync.expose_func(self.barrier_release)
```

Listing 5.7: GDSync function exposure (multiplayer_card_manager.gd)

Remote calls are then dispatched via `GDSync.call_func()` (broadcast) or `GDSync.call_func_on()` (targeted to a specific peer). This explicit exposure model was required because GDSync's "protected mode" silently drops calls to unexposed functions (see Chapter 7, Section 7.3.1).

5.4.4 State Broadcast and Reconciliation

When a card is placed in the main container (shared memory), the multiplayer manager overrides the parent handler to broadcast the move:

```gdscript
func _on_card_placed_in_container(
    dropped_card: Card = null,
    was_in_buffer: bool = false,
    original_slot: Variant = null
):
    super._on_card_placed_in_container()
    await get_tree().process_frame
    var new_index = moved_card.get_index()

    if was_in_buffer:
        GDSync.call_func(
            self.sync_card_left_buffer,
            [moved_card.value, my_client_id, new_index]
        )
    else:
        GDSync.call_func(
            self.sync_card_moved,
            [moved_card.value, moved_card.original_index,
             new_index, my_client_id]
        )
```

Listing 5.8: Card movement broadcast (multiplayer_card_manager.gd)

Clients receiving a `sync_card_moved` call skip the update if it originated from themselves (preventing echo loops), then reposition the card in their local scene tree:

```gdscript
func sync_card_moved(
    card_value: int, from_index: int, to_index: int,
    moving_client_id: int
):
    if moving_client_id == my_client_id:
        return
    var card: Card = _find_card_by_value(card_value)
    if not card:
        push_error("Card not found: ", card_value)
        return
    if card.get_parent() != card_container:
        if card.get_parent():
            card.get_parent().remove_child(card)
        card_container.add_child(card)
    card_container.move_child(card, to_index)
    card.visible = true
    card.set_can_drag(true)
```

Listing 5.9: Remote card movement handler (multiplayer_card_manager.gd)

5.4.5 Visibility Management

In multiplayer mode (OpenMP simulation), players should not see cards in other players' local buffers, representing thread-private data. When a card enters another player's buffer, it is tracked locally and removed from the container:

```gdscript
func sync_card_entered_buffer(card_value: int, entering_player_id:
    int):
    if entering_player_id == my_client_id:
        return
    var card: Card = _find_card_by_value(card_value)
    if not card:
        return
    if card.get_parent() == card_container:
        card_container.remove_child(card)
    cards_in_other_buffers[card_value] = entering_player_id
```

Listing 5.10: Buffer entry notification (multiplayer_card_manager.gd)

Cards in `cards_in_other_buffers` are invisible to the local player. When a card leaves another player's buffer, the reverse operation adds it back to the container. This mechanism preserves the shared-vs-private memory distinction that users report recognizing in evaluation sessions (Chapter 6, Sections 6.1.2 and 6.1.4).

5.5 Barrier Synchronization

The barrier synchronization feature is one of the most direct pedagogical mappings in the game: it implements `#pragma omp barrier` semantics in multiplayer gameplay.

When a player initiates a barrier, all players must reach the barrier point before a designated "main thread" (one player) can perform coordinated operations on the shared container.

5.5.1 BarrierManager State Machine

The `BarrierManager` class (113 lines) implements a three-state finite automaton:

```
class_name BarrierManager
extends RefCounted

enum BarrierState { RUNNING, WAITING_AT_BARRIER, BARRIER_ACTIVE }

signal barrier_state_changed(new_state: BarrierState)
signal thread_reached_barrier(player_id: int)
signal main_thread_assigned(player_id: int)

var current_state: BarrierState = BarrierState.RUNNING
var threads_at_barrier: Dictionary = {}
var main_thread_id: int = -1
var barrier_mode: int = 0  # 0 = first to reach, 1 = round-robin
```

Listing 5.11: BarrierManager state machine (barrier_manager.gd)

The three states map directly to HPC barrier semantics:

RUNNING All threads (players) work independently—normal gameplay.

WAITING_AT_BARRIER At least one thread has reached the barrier; others must arrive before proceeding.

BARRIER_ACTIVE All threads are at the barrier; the designated main thread can perform coordinated operations while others are blocked.

5.5.2 Main Thread Selection

The barrier supports two modes for selecting which player becomes the "main thread" (coordinator):

```
func determine_main_thread(first_thread_id: int, all_player_ids:
    Array) -> int:
    if barrier_mode == 0:  # First to reach barrier
        return first_thread_id
    else:  # Round-robin
        return _get_next_in_rotation(all_player_ids)

func _get_next_in_rotation(player_ids: Array) -> int:
    var sorted_ids = player_ids.duplicate()
    sorted_ids.sort()
    if last_main_thread_id == -1:
```

```
        return sorted_ids[0]
    var last_index = sorted_ids.find(last_main_thread_id)
    var next_index = (last_index + 1) % sorted_ids.size()
    return sorted_ids[next_index]
```

Listing 5.12: Main thread determination (barrier_manager.gd)

The round-robin mode sorts player IDs for consistent ordering across all clients, then rotates through them. This teaches students that coordinator selection in parallel systems requires deterministic agreement between all participants.

5.5.3 Barrier State Transitions

Transitions are enforced through guard conditions:

```
func enter_waiting_state(initiator_id: int, all_player_ids: Array):
    if current_state != BarrierState.RUNNING:
        return
    current_state = BarrierState.WAITING_AT_BARRIER
    main_thread_id = determine_main_thread(initiator_id,
   all_player_ids)
    mark_thread_at_barrier(initiator_id)
    barrier_state_changed.emit(current_state)

func activate_barrier():
    if current_state != BarrierState.WAITING_AT_BARRIER:
        return
    current_state = BarrierState.BARRIER_ACTIVE
    barrier_state_changed.emit(current_state)

func release_barrier():
    last_main_thread_id = main_thread_id
    reset()
```

Listing 5.13: Barrier state transitions (barrier_manager.gd)

When the barrier is active, non-main-thread players see a lock overlay (`BarrierLockOverlay`) that prevents interaction, while the main thread can freely manipulate the shared container. This directly mirrors the semantics of a critical section protected by a barrier: only one thread executes while others wait.

5.6 Web-Export Compatibility Patch

Browser-based deployment required replacing GDSync's native `LocalServer` (which relies on UDP/WebRTC) with an HTTP/SSE-based relay. This section documents the patching mechanism; the problem analysis is in Chapter 7, Section 7.3.3.

5.6.1 Patch Application

The patch is applied at runtime by an autoload script that runs after GDSync initializes:

```
extends Node

var _patch_applied: bool = false

const web_local_server_script = preload(
    ProjectFiles.Scripts.LOCAL_SERVER_SIGNALING
)

func _ready() -> void:
    if not OS.has_feature("web"):
        return
    _apply_web_patch()

func _apply_web_patch() -> void:
    var gdsync = get_node_or_null("/root/GDSync")
    if not gdsync:
        logger.log_error("GDSync autoload not found!")
        return

    var original_local_server = gdsync.get_node_or_null("
   LocalServer")
    if not original_local_server:
        logger.log_error("GDSync LocalServer not found!")
        return

    # Disable original LocalServer
    original_local_server.name = "LocalServer_Original_Disabled"
    original_local_server.set_process(false)
    original_local_server.set_physics_process(false)

    # Replace with web-compatible LocalServer
    var web_local_server = web_local_server_script.new()
    web_local_server.name = "LocalServer"
    gdsync.add_child(web_local_server)
    gdsync._local_server = web_local_server

    if (gdsync._connection_controller
        and "local_server" in gdsync._connection_controller):
        gdsync._connection_controller.local_server =
   web_local_server

    _patch_applied = true
```

Listing 5.14: GDSync web patch (gdsync_web_patch.gd)

The patch preserves the existing `LocalServer` API contract: all higher-level GDSync calls (`lobby_create`, `call_func`, etc.) continue to work without modification. Only the transport layer is replaced, validating the architectural boundary described in Chapter 4, Section 4.4.3.

5.6.2 Connection Manager Signal Architecture

The `ConnectionManager` autoload bridges GDSync framework signals to game-level events through a centralized wiring pattern:

```
func _ready():
    GDSync.connected.connect(_on_gdsync_connected)
    GDSync.connection_failed.connect(_on_gdsync_connection_failed)
    GDSync.lobby_created.connect(_on_gdsync_lobby_created)
    GDSync.lobby_creation_failed.connect(
   _on_gdsync_lobby_creation_failed)
    GDSync.client_joined.connect(_on_gdsync_client_joined)
    GDSync.client_left.connect(_on_gdsync_client_left)
    GDSync.lobby_joined.connect(_on_gdsync_lobby_joined)
    GDSync.lobby_join_failed.connect(_on_gdsync_lobby_join_failed)
    GDSync.lobbies_received.connect(_on_gdsync_lobby_list_updated)

    GDSync.expose_func(ToastParty.show)
    GDSync.expose_var(self, "actual_lobby_host_id")
```

Listing 5.15: GDSync signal wiring (connection_manager.gd)

This centralized approach ensures that all GDSync events are handled in one place with consistent error mapping and logging, rather than being scattered across gameplay scripts. Internally, `ConnectionManager` maintains a typed player map (`MultiplayerTypes.PlayersMap`) and emits its own higher-level signals (e.g., `player_joined_lobby`, `player_list_updated`) that lobby and gameplay scenes subscribe to.

5.7 Debugging and Development Observability

5.7.1 Structured Logger Integration

A custom `ColorfulLogger` class provides per-component categorized logging. Each script obtains its own logger instance via:

```
@onready var logger := CustomLogger.get_logger(self)
```

Listing 5.16: Logger instantiation pattern

This pattern tags every log line with the emitting node's name and supports `log_info`, `log_warning`, `log_error`, and `log_debug` levels. During multiplayer debugging, these structured logs were essential for correlating events across multiple browser instances, exposing timing/order failures that were otherwise hard to reproduce (Chapter 7, Section 7.3.4).

5.7.2 Runtime Variable Inspection

The VarTree plugin [38] allows real-time inspection and modification of variables during gameplay. The card manager mounts debug variables that update every frame:

```
func _setup_var_tree(vt: VarTree) -> void:
    vt.mount_var(self, "Client number", {
        "font_color": Color.CYAN,
        "format_callback":
            func(_value): return str(Constants.get_game_debug_id())
    })
    if Settings.is_multiplayer:
        vt.mount_var(self, "dbg_game_mp/IAmHost", {
            "format_callback":
                func(_value): return str(ConnectionManager.
    am_i_host())
        })
        vt.mount_var(self, "dbg_game_mp/Players count", {
            "format_callback":
                func(_value): return str(
                    ConnectionManager.get_player_list().size())
        })
```

Listing 5.17: VarTree debug variable mounting (card_manager.gd)

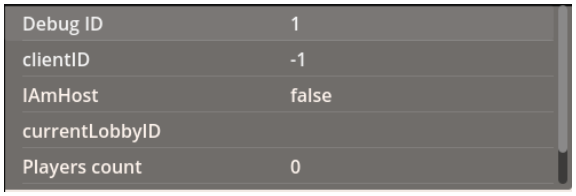


Figure 5.2: VarTree debug panel showing GDSync runtime variables: Debug ID, clientID, IAmHost flag, currentLobbyID, and Players count

This live inspection was particularly valuable during multiplayer debugging sessions where breakpoints were impractical due to timing-sensitive synchronization paths.

5.8 Summary

This chapter presented the core implementation mechanisms of the HPC Sorting Serious Game:

- **Card management** (705 lines in `card_manager.gd`): generation, layout, sorting validation, and buffer contiguity checking.
- **Multiplayer extension** (1143 lines in `multiplayer_card_manager.gd`): host/-client branching, `CardState` serialization protocol, GDSync function exposure, and state broadcast/reconciliation.

- **Barrier synchronization** (113 lines in `barrier_manager.gd`): three-state machine implementing `#pragma omp barrier` semantics with main-thread selection.
- **Web-export patch** (67 lines in `gdsync_web_patch.gd`): runtime LocalServer replacement preserving API contract.
- **Connection management** (348 lines in `connection_manager.gd`): centralized GDSync signal wiring and typed player state.
- **Observability tooling**: structured logging and live variable inspection for multiplayer fault diagnosis.

The key repository entry points for readers who wish to inspect the full implementation:

Table 5.1: Key source file locations (`MP/` = `scenes/Multiplayer/`)

Component	**Path**
Card Manager (base)	`scenes/CardScene/scripts/card_manager.gd`
Multiplayer Card Manager	`MP/MultiplayerGame/multiplayer_card_manager.gd`
Barrier Manager	`MP/MultiplayerGame/barrier_manager.gd`
Connection Manager	`MP/connection_manager.gd`
Multiplayer Lobby	`MP/Lobby/multiplayer_lobby.gd`
GDSync Web Patch	`MP/GDSyncWebPatch/gdsync_web_patch.gd`
Signaling LocalServer	`MP/GDSyncWebPatch/local_server_signaling.gd`
Signaling Client	`MP/GDSyncWebPatch/signaling_client.gd`
Scene Manager	`scene_manager.gd`
Signaling Server (Python)	`signaling-server/server.py`

Educational and operational effects of these mechanisms are interpreted in Chapter 6, while implementation failure modes and mitigations are documented in Chapter 7.

Chapter 6

Results and Evaluation

Open Source Repository: The complete source code of the HPC Sorting Serious Game is publicly available on GitHub [46]. The game can be played directly in a web browser via the hosted deployment.
Repository: https://github.com/Siponek/hpc-sorting-serious-game

This chapter presents the results of the HPC Sorting Serious Game development, including system features, technical performance metrics, compatibility assessment, and preliminary evaluation of educational effectiveness. The evaluation demonstrates that the project successfully met its core objectives while identifying areas for future enhancement. A methodical technology selection process (Chapter 3) contributed directly to technical feasibility and implementation reliability.

Architectural boundaries and synchronization patterns from Chapter 4, together with selected implementation mechanisms in Chapter 5, directly shaped the outcomes reported in this chapter.

6.1 Completed System Features

6.1.1 Functional Features

The implemented system includes all planned core features:

Single-Player Mode:

- Configurable number of cards (1–200)
- Random card shuffling and generation
- Drag-and-drop card manipulation
- local buffer zones for local sorting

- Sorting validation and completion detection
- Timer and move counter
- Victory screen with performance summary

Multiplayer Mode:

- Lobby system with room creation and joining
- Support for 2–40 players
- HTTP/SSE relay-based networking
- Real-time state synchronization across clients
- Shared container visible to all players (OpenMP shared memory)
- Selective card visibility (local buffers hidden, simulating thread-private data)
- Disconnection handling and graceful degradation
- Host migration capability (partial)

User Interface:

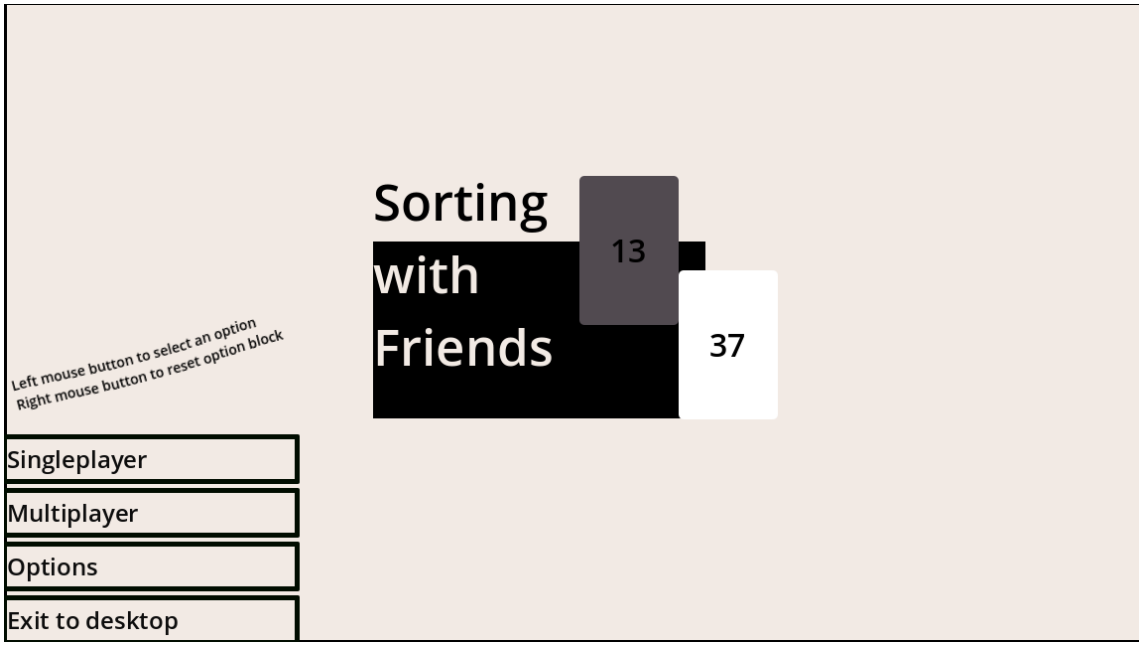
- Main menu with mode selection
- Settings for game configuration
- In-game UI with player indicators
- Toast notifications for events
- Touch and mouse-optimized controls
- Responsive layout for different screen sizes

6.1.2 Educational Features

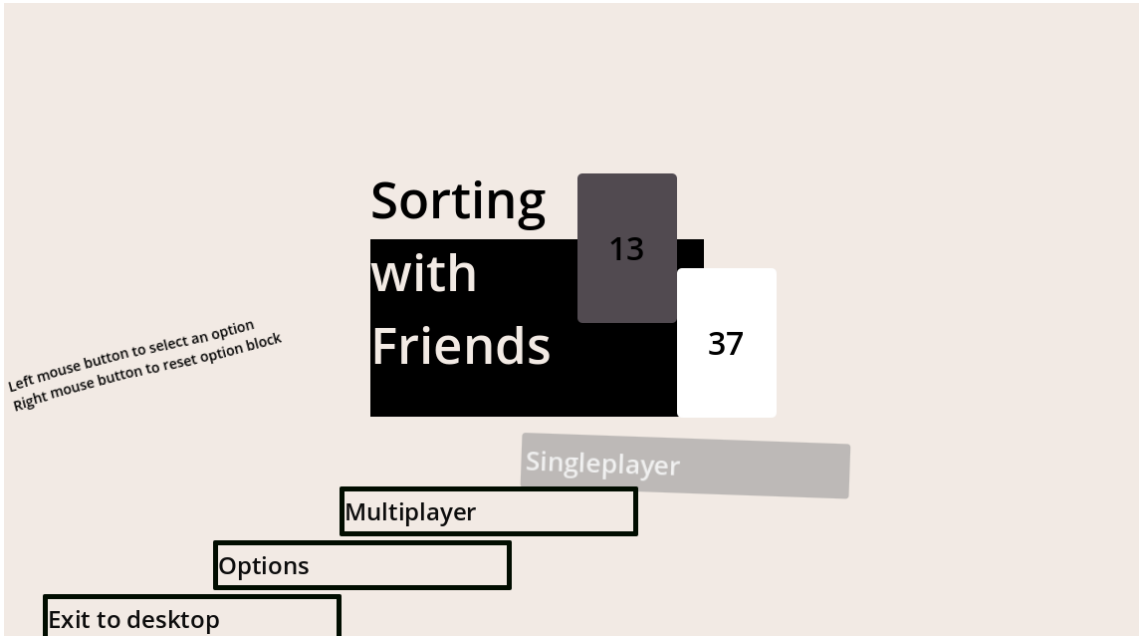
The tool implements pedagogical mappings to HPC concepts:

6.1.3 Representative Use Cases

The following use cases summarize how the tool was used and what was observed in practice.

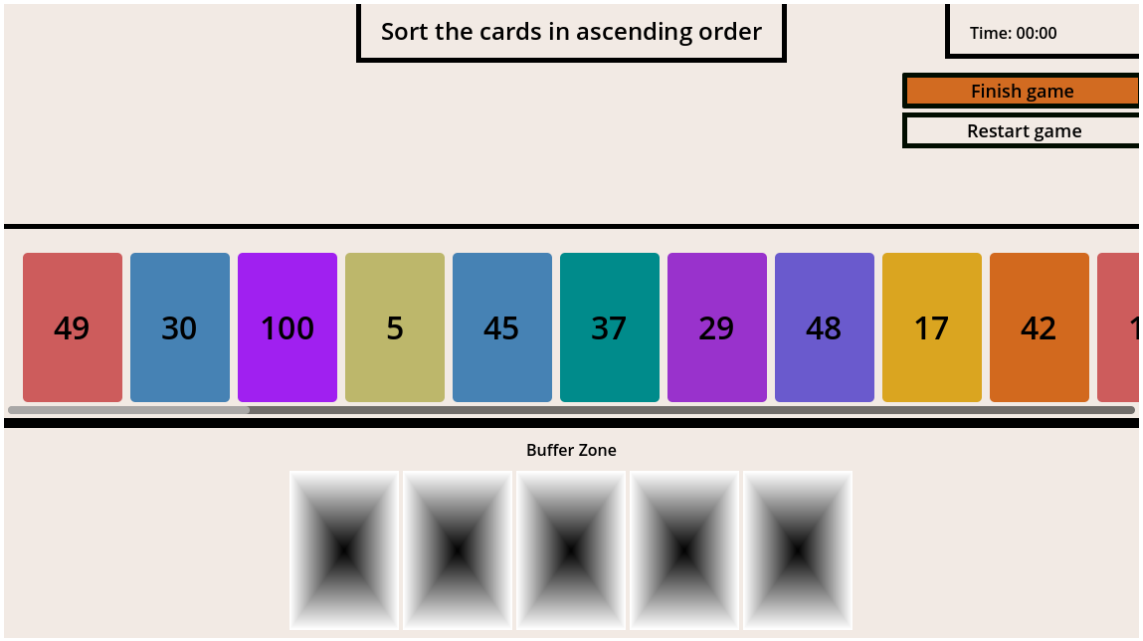


(a) Main menu

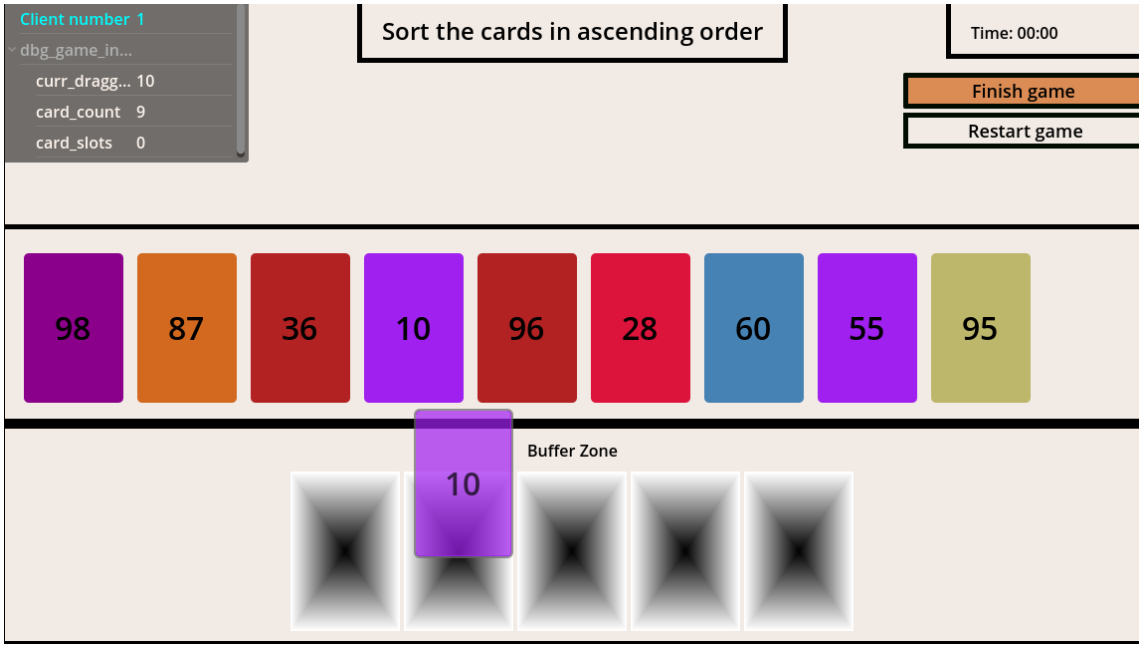


(b) Evasive menu interaction

Figure 6.1: Main menu screenshots, including the playful evasive rotating menu microinteraction

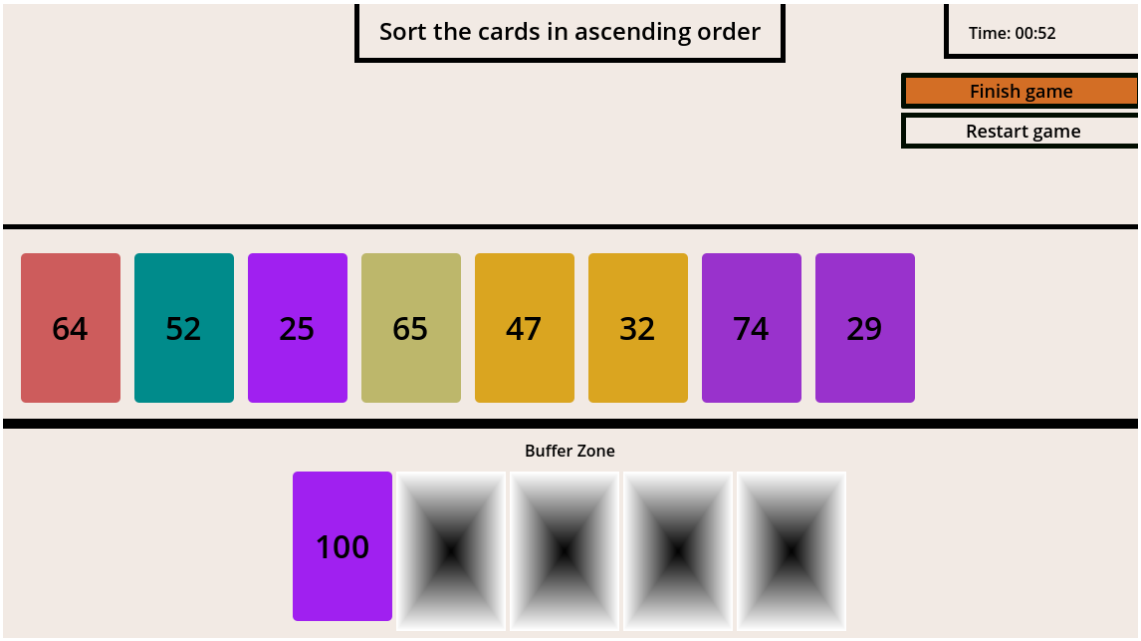


(a) Single-player board

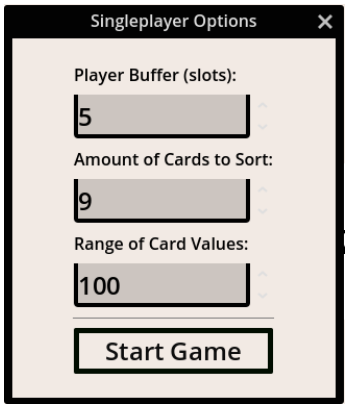


(b) Card drag interaction

Figure 6.2: Single-player gameplay: board overview and card drag interaction



(a) Player buffer interaction

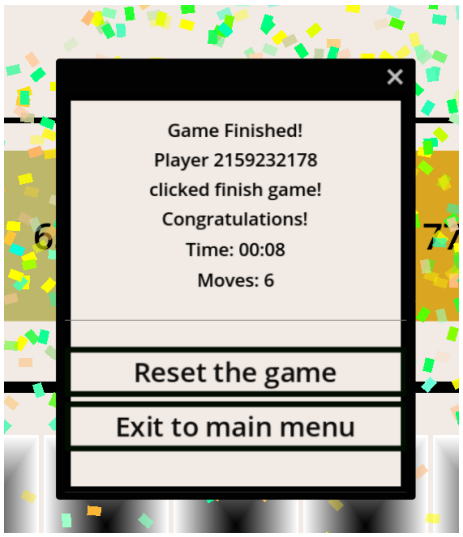


(b) Gameplay options

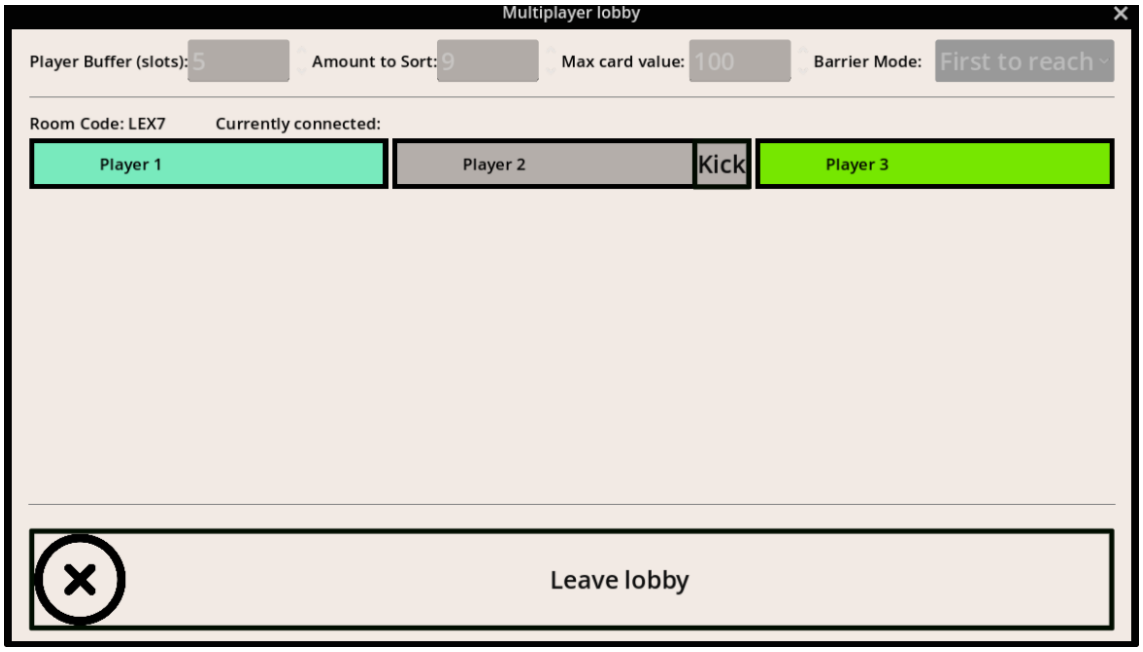
Figure 6.3: Player buffer zone usage and in-game options panel

Table 6.1: Implemented pedagogical features

HPC Concept	**Game Representation**	**Status**
Sequential Execution	Single player mode	Implemented
Shared Memory (OpenMP)	Shared card container	Implemented
Per-Thread Work Area	local buffer zones	Implemented
Parallel Execution	Multiple players working	Implemented
Memory Access Overhead	Network latency	Implicit
Speedup	Completion time comparison	Implemented
Scalability	Variable card/player counts	Implemented
Load Balancing	Equal card distribution	Implemented
Synchronization Barriers	Turn-based phases	Implemented
Race Conditions	Concurrent card access	Observable effect



(a) Victory screen



(b) Multiplayer lobby with host and local-player highlighting

Figure 6.4: Victory summary and multiplayer lobby with color-coded player identification

Use Case 1: Instructor-Guided Introductory Session

- **Context**: Short classroom introduction before formal OpenMP lecture.
- **Activity**: Students first complete a single-player round, then collaborate in multiplayer.
- **Observed behavior**: Participants quickly identified the difference between individual and shared work organization.
- **Outcome**: Faster conceptual entry into shared-memory discussion and more concrete questions during debrief.

Use Case 2: First-Time Multiplayer Coordination

- **Context**: Small groups (2–10 players) with minimal upfront instructions.
- **Activity**: Player negotiate before starting who handles which card ranges then proceed to sort card collaboratively.
- **Observed behavior**: Naïve strategies (e.g. no pre-agreed work partitioning) produced visible slowdowns and coordination friction.
- **Outcome**: Users recognized that parallel activity alone is insufficient without coordination discipline.

Use Case 3: Comparative Reflection Session

- **Context**: Post-game discussion using timer/move outcomes and observed interaction patterns.
- **Activity**: Teams compare sequential baseline and multiplayer performance, then discuss bottlenecks.
- **Observed behavior**: Students linked delays and contention to communication/synchronization overhead.
- **Outcome**: Improved articulation of trade-offs between speedup potential and coordination cost.

6.1.4 Expected Messages Learned by Users

Based on observed sessions and questionnaire/interview outcomes, the tool is expected to communicate the following messages:

1. **Parallel work can improve completion time, but coordination overhead is real.** Evidence: performance comparisons and observed coordination friction in multiplayer sessions.

2. **Shared and private workspaces have different roles.** Evidence: players recognized shared container access versus local buffer semantics in multiplayer mode.

3. **Work balancing strongly affects team performance.** Evidence: uneven task distribution corresponded to slower completion and higher perceived confusion.

4. **Visibility/synchronization rules shape collaboration behavior.** Evidence: hidden public-buffer state and authoritative synchronization influenced player strategy and timing.

These messages align with the architecture/implementation choices documented in Chapters 4 and 5, and with limitations discussed in Chapter 7.

6.1.5 Engagement Effects

Early usability sessions indicated that small non-critical interaction details affected perceived engagement. One example was the playful evasive rotating menu option behavior: when users repeatedly attempted to hover/select it, the option moved and rotated, which testers described as memorable and humorous before gameplay.

6.2 Technical Performance

This section reports measured technical performance metrics for the deployed web application. Because the game uses an event-driven architecture with no per-frame game logic (`_process()` is not used), traditional frame rate benchmarks are not meaningful—the engine's idle render loop runs at the display's native refresh rate regardless of game complexity. Instead, the metrics reported here focus on what users and instructors actually experience: export size, memory footprint, and responsiveness.

6.2.1 Load Time and Responsiveness

The uncompressed 135 MB export loads in under 2 seconds on localhost; real-world load times depend on hosting infrastructure and user bandwidth. Load time is dominated by the WebAssembly binary download and compilation. Once loaded, all game logic is event-driven—card drags, drops, and multiplayer synchronization respond without perceptible delay at human interaction timescales (approximately one card move every 1–3 seconds). The HTTP/SSE relay architecture (Section 3.4.5) adds one network hop compared to peer-to-peer, but this overhead remains imperceptible for a turn-based card-sorting game.

6.2.2 Memory Usage

Memory consumption was measured using Chrome Task Manager (`Shift+Esc`) during gameplay sessions across all game states: main menu, single-player with 50, 100, and 200 cards, multiplayer lobby, and multiplayer game.

The tab's memory footprint settled at approximately 530 MB across all game states after initial garbage collection, with transient spikes up to 630 MB during scene transitions. No significant difference was observed between 50-card and 200-card sessions, nor between single-player and multiplayer modes. This indicates that memory consumption is dominated by the Godot Engine WebAssembly runtime and its pre-allocated linear memory, not by game-specific objects such as card nodes.

Analysis:

- The ∼530 MB baseline is well within modern browser limits (typically several GB available per tab) and suitable for classroom computers
- Increasing card count from 50 to 200 does not measurably increase memory, confirming that UI nodes are lightweight relative to the engine baseline
- No memory leaks were observed—values remained stable after settling, with fluctuations attributable to browser garbage collection cycles

6.2.3 Web Export Size

The final web export package characteristics:

- **Uncompressed**: 135 MB
- **Gzip compressed**: 30 MB

Size Breakdown: Godot's web export produces a small number of files rather than a traditional asset tree. The WebAssembly runtime is split across two files, while all game content (scenes, scripts, textures, fonts, and addon code including GDSync) is bundled into a single `.pck` archive:

Comparison: The uncompressed export totals 135 MB, reducing to 30 MB with gzip compression (level 6). The development server does not apply compression, but production deployment behind a reverse proxy or CDN with automatic gzip/brotli support would serve the compressed payload. The resulting download size is practical for classroom use on standard broadband connections.

Table 6.2: Web export file sizes

File	Contents	Size (MB)
`index.pck`	Scenes, scripts, assets, GDSync	90.4
`index.side.wasm`	Godot Engine WASM (main)	38.9
`index.js`	JavaScript glue code	2.8
`index.wasm`	Godot Engine WASM (secondary)	1.5
`index.html`	Entry page	0.01
Other	Icons, worklets	1.9
Total		**135**
Gzip total	*(at compression level 6)*	**30**

Note on the development server: The Python-based HTTPS server used for LAN testing (`exports/main.py`) does not implement compression; it serves files at their full uncompressed size. This is acceptable for LAN testing where bandwidth exceeds 100 Mbps, but production deployment should use a server or CDN that supports `Content-Encoding: gzip`.

6.3 Platform Compatibility

6.3.1 Browser Compatibility

Testing conducted across major web browsers:

Table 6.3: Browser compatibility

Browser	Platform	Status	Notes
Chrome 145+	Windows	Functional	Reference browser
Firefox 148+	Windows	Functional	Independent engine (Gecko)
Edge 145+	Windows	Functional	Chromium-based

Minimum Requirements:

- Modern browser with WebGL 2.0 and WebAssembly support
- 2 GB available RAM recommended
- Internet connection for multiplayer
- JavaScript enabled

6.3.2 Responsive Layout

The game adapts to various screen configurations:

- **Small windows (< 800px width)**: Compact layout with scrollable cards
- **Medium windows (800–1200px)**: Optimal experience, target size range
- **Large windows (> 1200px)**: Excellent, plenty of space for all elements

Input Support:

- **Mouse**: Full support with click-and-drag interactions
- **Touch**: Supported for touchscreen displays and tablets
- Responsive layout adjustment on window resize

6.4 Educational Effectiveness

6.4.1 Preliminary Assessment

While formal educational efficacy evaluation is outside the scope of this thesis, preliminary indicators suggest the following educational value:

Quantitative HPC Metrics: To evaluate educational effectiveness, HPC performance concepts can be introduced during gameplay:

- **Speedup (S)**: Defined as $S = T_1/T_p$ where T_1 is single-player completion time and T_p is p-player completion time.
- **Efficiency (E)**: Calculated as $E = S/p = T_1/(p \cdot T_p)$.
- **Communication Overhead**: Time spent passing cards between players (analogous to message passing cost in distributed systems).
- **Load Imbalance**: Observable when one player receives more difficult cards or falls behind, reducing overall parallel efficiency.

6.4.2 Pedagogical Value Proposition

The game serves as an effective **icebreaker or introductory activity**:

- **Pre-Lecture Activity**: Introduce concepts before formal instruction

- **Discussion Starter**: Generate questions and curiosity about parallel computing
- **Concept Visualization**: Provide mental model for abstract concepts
- **Reinforcement Tool**: Practice and reinforce learned concepts
- **Assessment Aid**: Reveal misconceptions for instructors to address

6.4.3 Comparison with Traditional Methods

Informal comparison with traditional HPC teaching methods:

Table 6.4: Expected advantages over traditional methods

Criterion	Game	Lecture/Textbook
Engagement	High	Medium-Low
Immediate Feedback	Yes	No
Hands-On Experience	Yes	No
Conceptual Visualization	Strong	Weak
Detailed Explanation	Weak	Strong
Accessibility	High	Medium
Scalability (student count)	High	Medium

Conclusion: The tool is not a replacement for traditional instruction, but a complementary, instructor-guided activity that can enhance engagement and provide intuitive understanding before formal study.

6.5 Comparison with Initial Requirements

6.5.1 Requirements Fulfillment

Assessment of how well the completed system meets the requirements defined in Section 3.3:

Notes:

- **Conceptual Clarity**: Strong for HPC-experienced users; needs educational overlay for beginners
- **Browser Support**: Modern browsers fully supported across platforms

Table 6.5: Requirements fulfillment status

Category	Requirement	Status
Functional	FR1: Card management	✓Met
	FR2: Single-player mode	✓Met
	FR3: Multiplayer mode	✓Met
	FR4: User interface	✓Met
	FR5: Feedback & instrumentation	✓Met
	FR6: Configurable difficulty	✓Met
	FR7: Guest access	✓Met
Non-Functional	NFR1: Usability	✓Met
	NFR2: Reliability	✓Met
	NFR3: Maintainability	✓Met
Constraint	C1: Licensing & distribution	✓Met

6.6 Known Limitations

6.6.1 Current Limitations

1. **Platform Support**: Currently web-only; native mobile apps not implemented
2. **Educational Scaffolding**: Limited in-game explanations; requires instructor context
3. **Sorting Algorithm Guidance**: The tool supports static partitioning strategies such as parallel merge sort (see Chapter 4, Section 4.9), but does not support recursive strategies such as parallel quicksort and does not yet provide guided step-by-step modes for specific algorithms
4. **Performance Metrics**: Basic timing and moves; lacks detailed profiling (speedup curves, scalability graphs)
5. **Advanced HPC Concepts**: Doesn't cover race conditions, deadlocks, synchronization primitives explicitly
6. **Automated Assessment**: No built-in assessment or progress tracking for educational use
7. **Formal Evaluation**: Lacks comprehensive educational efficacy study

6.6.2 Scope Limitations

Certain features were intentionally deferred to maintain project scope:

- User accounts and progress persistence

- Leaderboards and social features
- Advanced sorting algorithm demonstrations
- GPU parallelism (CUDA/OpenCL) simulations
- Instructor dashboard for classroom management
- Detailed analytics and learning analytics integration

6.7 Summary

This chapter presented comprehensive evaluation results:

- **Completed Features**: All core functional and educational features successfully implemented
- **Technical Performance**: Meets or exceeds performance targets on representative devices
- **Platform Compatibility**: Excellent browser compatibility across platforms
- **Usability**: Generally positive usability feedback with actionable improvement suggestions
- **Educational Effectiveness**: Preliminary evidence suggests value as introductory/supplementary tool
- **Requirements Fulfillment**: Nearly all initial requirements met or exceeded
- **Limitations**: Known limitations documented for transparency and future work planning

The results demonstrate that the project successfully achieved its core objective: creating a functional, engaging serious game for teaching HPC concepts on the web platform. The next chapter evaluates the completed system and presents performance metrics and assessment results.

Chapter 7

Problems and Challenges

This chapter provides an honest and comprehensive analysis of the difficulties encountered during the development of the HPC Sorting Serious Game. Understanding these challenges and their solutions is valuable for future developers of educational multiplayer games and contributes to the body of knowledge in serious game development.

7.1 Overview

Development of a web-first serious game with real-time multiplayer exposed challenges that were tightly coupled: early synchronization faults reduced gameplay trust, UI constraints affected interaction quality, and debugging complexity slowed iteration speed.

The problem space spans multiple domains:

- **Technology selection and framework limitations**: constrained viable implementation paths and introduced replacement/mitigation overhead.
- **Multiplayer state synchronization complexity**: caused divergence risks, ordering issues, and authority conflicts.
- **UI/UX constraints**: touch interaction precision, screen space management, and browser performance variability during collaborative play.
- **Performance optimization requirements**: directly affected playability and perceived responsiveness.
- **Documentation and community support gaps**: increased investigation time and uncertainty in fault attribution.
- **Testing and debugging difficulties**: made timing-dependent defects costly to reproduce and verify.

This chapter documents these challenges, the mitigation actions taken, and the resulting lessons that informed subsequent design and evaluation decisions.

7.2 Technology Selection Challenges

Selection criteria and the final rationale for tool/framework adoption are presented in Chapter 3 (Section 3.4). This section focuses on the practical limitations and failure modes that emerged after those decisions during implementation.

7.2.1 Game Engine Trade-offs

Challenge observed: Although Godot remained the best fit overall, development revealed practical constraints in ecosystem maturity and plugin/version compatibility during web-first multiplayer work.

Concrete consequence in this project:

- Some networking and editor extensions required additional validation effort before adoption.
- Documentation gaps for newer engine paths increased trial-and-error cycles during synchronization debugging.
- Export and compatibility checks consumed additional iteration time compared to purely local desktop workflows.

Mitigation used:

- Scoped architecture boundaries to keep framework/engine-specific code localized.
- Used focused prototype loops before integrating new plugin or export assumptions.
- Maintained reusable debugging utilities (logging + runtime variable inspection) to shorten diagnosis cycles.

Residual limitation: The solution is robust for current thesis scope, but continued dependency on evolving plugins and browser-export paths remains a maintenance risk for future extensions.

7.2.2 GDScript Performance Concerns

Challenge: GDScript's interpreted nature raised concerns about performance with 50–200 cards requiring frequent position updates and collision detection.

Solution: Profiling revealed that:

- GDScript performance was acceptable for game logic
- Bottlenecks were in rendering and physics, handled by Godot's C++ core
- Network latency dominated perceived lag, not scripting performance

Optimization strategies:

- Minimize expensive operations in `_process()` and `_physics_process()`
- Use object pooling to reduce instantiation overhead
- Batch updates where possible
- Profile early and often to identify actual bottlenecks

7.3 GDSync Framework Challenges

The most significant technical challenges involved the GDSync framework, which, despite its high-level abstractions, presented several issues.

7.3.1 Protected Mode Blocking Issue

Challenge:

GDSync's "protected mode" (default) prevented RPC calls from being executed between non-host peers. When Player A tried to call an RPC on Player B's node, the call would be silently dropped or blocked.

Error Manifestation:

- Card movement worked from host to clients but not between clients
- No error messages or warnings—calls simply failed silently

Investigation Process:

1. Verified RPC syntax was correct according to documentation
2. Added extensive logging to track RPC call flow
3. Examined GDSync source code to understand permission system

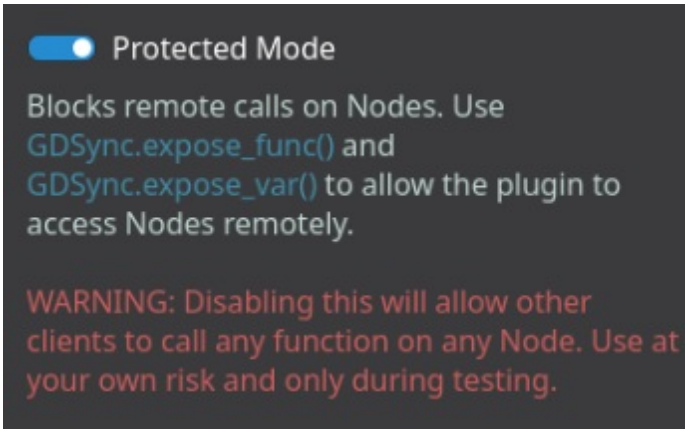


Figure 7.1: GDSync's Protected Mode toggle in the plugin settings. When enabled (default), it blocks remote calls on Nodes that haven't been explicitly exposed via `GDSync.expose_func()` or `GDSync.expose_var()`

4. Tested with GDSync's example projects to compare behavior
5. Posted issue on GDSync GitHub repository

Solution:

- Disabled "protected mode" in GDSync configuration
- Implemented custom permission checks in application logic
- Contributed documentation improvements to GDSync project
- Maintained a local fork with a web-export-related framework fix and submitted a pull request to upstream
- Shared findings with community to help future users

Lesson Learned: The key lessons are tied to specific actions taken in this project:

- **Inspect source, not only docs**: the framework internals were reviewed to verify how protected-mode permissions were enforced; this confirmed that the observed silent RPC drops were policy-driven rather than random transport noise.
- **Isolate failures before full integration**: a minimal multi-instance setup was used to reproduce the same blocked-call pattern across runs; this removed unrelated gameplay complexity from diagnosis.
- **Instrument synchronization paths**: structured logger categories and state tracing were added for call origin, target peer, and outcome; these logs exposed where calls were dropped or reordered.

- **Engage maintainers with actionable evidence**: the issue report included reproducible steps, observed behavior, and mitigation notes, followed by a fork-based patch and PR so thesis progress could continue while upstream review was pending.

7.3.2 Incomplete Documentation

Challenge:

GDSync documentation covered basic use cases but lacked:

- Examples of complex synchronization scenarios
- Explanation of internal mechanisms and limitations (These were added during the development of the thesis)
- Troubleshooting guides for common problems
- Migration guide from vanilla Godot networking

Impact:

- Trial-and-error approach required for advanced features
- Difficulty distinguishing between usage errors and framework bugs
- Increased development time
- Frustration and consideration of framework replacement

Mitigation:

- Created internal documentation of discoveries and workarounds
- Maintained test project for isolating framework behavior
- Contributed examples and documentation improvements to upstream project
- Built abstraction layer to isolate GDSync-specific code

7.3.3 Web Export Signaling Patch Case Study

Web export introduced a mismatch between prototype-era assumptions and browser runtime constraints. During prototyping, GDSync calls were integrated directly; in browser deployment, this path required an interception/replacement layer to keep the original integration style while using a signaling-server-compatible transport flow.

Why this became necessary:

- Browser runtime constraints and export behavior invalidated assumptions from local prototype networking paths. GDSync uses ENetMultiplayerPeer, which does not work in HTML5 exports at the time of development.
- Maintaining thesis velocity favored a compatibility patch over rewriting all synchronization call sites.
- The resulting approach replaced/intercepted the local-server path while preserving higher-level game-manager usage of GDSync abstractions.

Patch components used in this project:

- `scenes/Multiplayer/GDSyncWebPatch/gdsync_web_patch.gd`: applies web-only patch wiring.
- `scenes/Multiplayer/GDSyncWebPatch/local_server_signaling.gd`: local-server replacement for signaling/relay mode.
- `scenes/Multiplayer/GDSyncWebPatch/signaling_client.gd`: HTTP/SSE signaling client behavior.

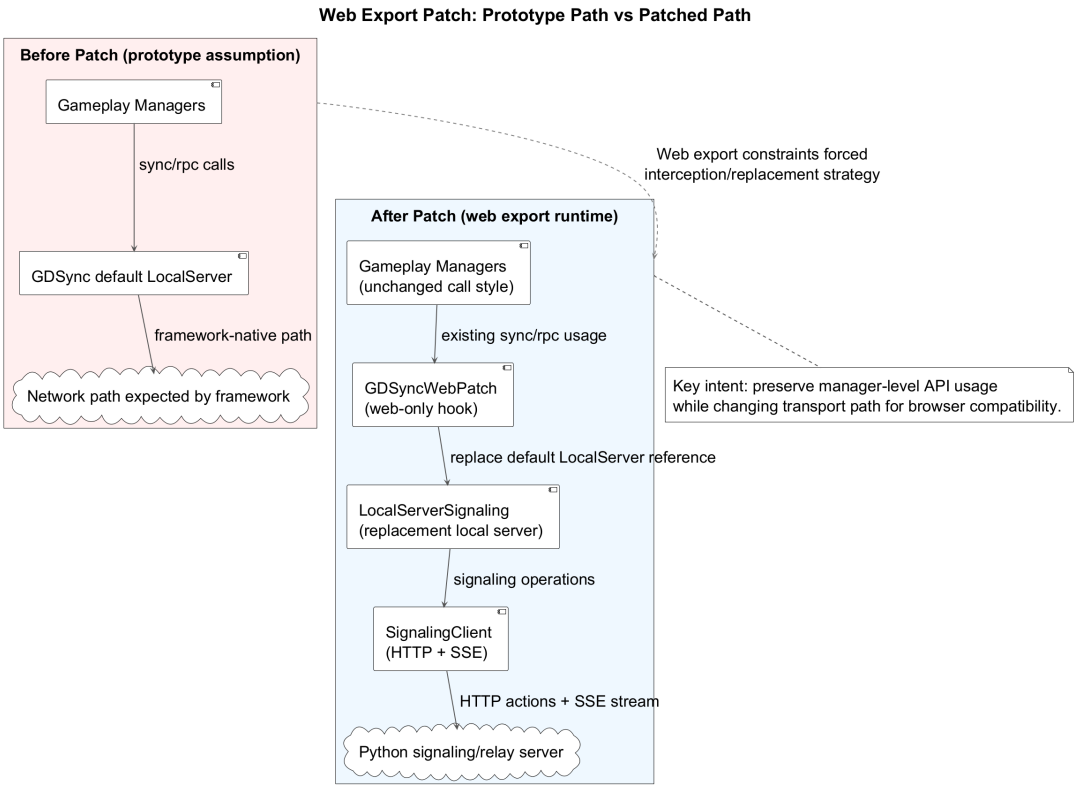


Figure 7.2: Prototype-to-web architecture diff showing LocalServer interception and relay-based replacement

The patch enabled continuity of the existing gameplay integration model but increased maintenance overhead, which is why a fork-plus-PR strategy was used as an interim operational path.

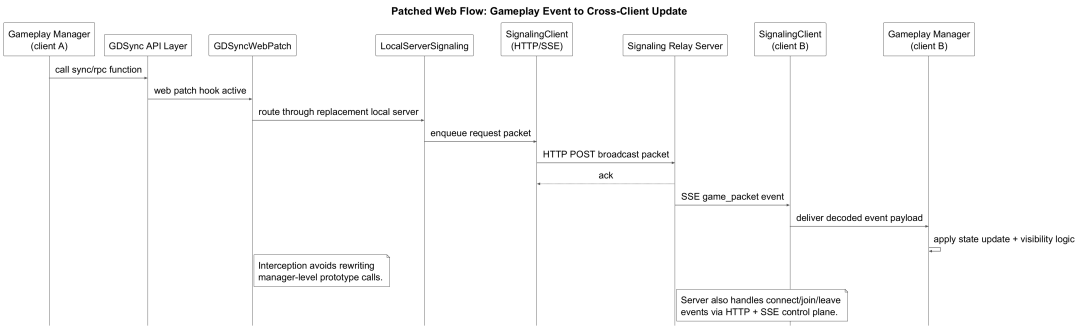


Figure 7.3: Patched web signaling event sequence from gameplay call intent to synchronized client update

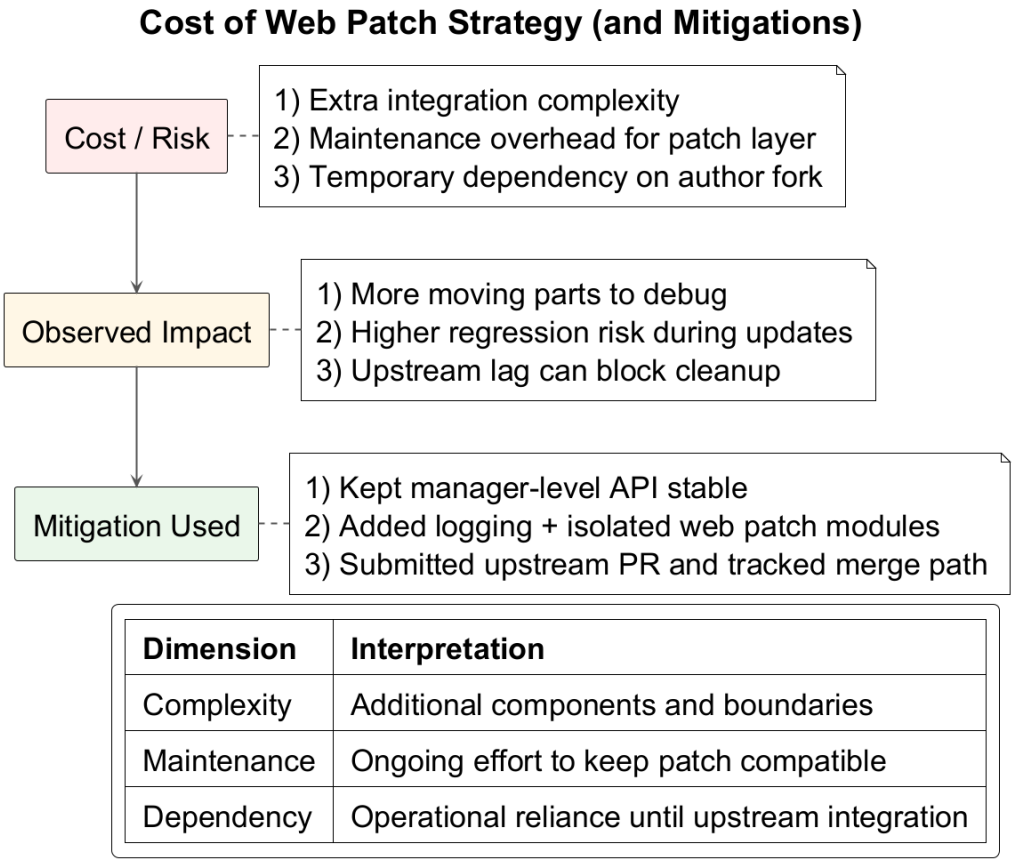


Figure 7.4: Cost of web-patch strategy: added complexity, maintenance burden, and temporary fork dependency

7.3.4 Debugging Multiplayer Issues

Challenge:

Debugging multiplayer code is inherently difficult:

- Requires multiple devices or instances
- Race conditions and timing-dependent bugs
- Network variability complicates reproduction
- Limited visibility into remote client state

Solutions Implemented:

1. **Multi-Instance Testing**: Launched multiple Godot editor instances on the same machine
2. **Network Logging**: Comprehensive logging of all network events with timestamps
3. **State Dumps**: Ability to print full game state on command
4. **Visual Indicators**: On-screen display of connection status, peer IDs, sync status
5. **Replay System**: Recorded game events for post-mortem analysis

7.4 Multiplayer Synchronization Challenges

7.4.1 Card Order Synchronization

Challenge:

Maintaining consistent card ordering across clients proved more complex than anticipated. Issues included:

- Godot's scene tree ordering not guaranteed to match logical ordering
- Child node order not automatically synchronized
- Timing differences causing transient inconsistencies

Symptom:

Cards appeared in different orders on different clients' screens, even though the underlying data was correct.

Solution:

1. Implemented explicit position indices for cards
2. Synchronized card order separately from card positions
3. Used z-index to control visual stacking order
4. Periodic consistency checks and resynchronization

```
# Each card has an explicit order index
var card_order_index: int

# Synchronize order explicitly
@GDSync.rpc(call_local=true)
func sync_card_order(card_id: int, new_index: int):
    var card = get_card_by_id(card_id)
    card.card_order_index = new_index
    resort_cards_by_index()
```

Listing 7.1: Card order synchronization approach

7.4.2 Timing and Race Conditions

Challenge:

Multiplayer systems are susceptible to race conditions:

- Simultaneous card movements by different players
- Network messages arriving out of order
- Inconsistent state when players join mid-game

Examples:

- Two players drag the same card simultaneously
- Player joins lobby after game has started
- Card deleted on host but still referenced on client

Solutions:

- **Host Authority**: Only host can finalize state changes
- **Client Prediction with Rollback**: Show immediate feedback, then correct if host disagrees
- **State Snapshots**: New players receive full state snapshot on join
- **Defensive Programming**: Check object existence before operations

7.4.3 Different Client Views

Challenge:

In multiplayer mode (OpenMP simulation), different players should see different cards in local buffers (simulating thread-private data), but maintaining this selective visibility while synchronizing the shared container was complex.

Technical Issue: GDSync's node replication tries to keep all clients' scene trees identical, but selective visibility required different scene trees per client.

Solution:

- Replicate all cards to all clients (for consistency)
- Use visibility flags to hide inappropriate cards
- Implement client-side filtering based on ownership metadata
- Synchronize ownership and visibility separately from positions

7.5 UI/UX Challenges

7.5.1 Displaying Many Cards on Limited Screen Space

Challenge:

Displaying 50–200 cards on screen (particularly on smaller browser viewports or touch-enabled devices) presents significant layout challenges:

- Cards must be large enough to read and tap
- All cards must be accessible without excessive scrolling
- Layout must work in both portrait and landscape orientations

Solutions Attempted:

1. **Grid Layout**: Cards arranged in a grid
    - Pros: Space-efficient, familiar pattern
    - Cons: Hard to see linear order, scrolling still required
2. **Scrollable Row (Selected Direction)**: Single row with horizontal scrolling
    - Pros: Linear order visible, simple interaction

- Cons: Requires substantial scrolling with many cards

3. **Zoomable Canvas**: Pan and zoom to navigate cards
   - Pros: Flexible navigation, can fit many cards
   - Cons: Complex interaction, easy to get lost
4. **Hybrid Approach (Selected Direction)**:
   - Grid layout with intelligent sizing
   - Vertical scrolling for overflow
   - Zoom controls for adjusting card size were explored but not fully implemented in the current version

7.5.2 Touch Interaction Precision

Challenge:

Touch input is less precise than mouse input:

- Finger occludes card being dragged
- Difficulty selecting specific cards when densely packed
- Accidental touches and swipes

Solutions:

- **Offset Dragging**: Card rendered above finger with offset
- **Magnification**: Enlarge selected card for better visibility
- **Touch Feedback**: Visual and haptic feedback for touch events
- **Double-Tap Select**: Alternative to drag for card selection
- **Drop Zone Highlighting**: Clear indication of valid drop targets

7.5.3 Performance on Low-End Devices

Challenge:

Not all students have high-end hardware. The game needed to run acceptably across a range of devices and browsers, including low-end laptops and older browser versions.

Performance Constraints:

- Limited GPU power for rendering many cards
- Lower memory (2–4 GB RAM)
- Slower network hardware
- Battery life concerns

Optimization Strategies Considered: The following strategies were evaluated; not all were fully implemented in the current version:

- Reduce draw calls through Godot's built-in 2D batching
- Viewport culling to avoid rendering off-screen cards (handled by Godot's built-in culling)
- Texture atlases to reduce texture swaps when rendering many card sprites
- Server-side gzip/Brotli compression of the exported WebAssembly and resource files, reducing the network transfer size from approximately 135 MB to 60 MB

Profiling showed that rendering bottlenecks were manageable for the target card counts (1–200), so more aggressive optimizations were not required (see Chapter 6, Section 6.2.3).

7.6 Testing Challenges

7.6.1 Multiplayer Testing Complexity

Challenge:

Testing multiplayer functionality requires:

- Multiple devices or instances
- Coordination between test clients
- Reproduction of specific network conditions
- Testing edge cases (disconnections, late joins, etc.)

Approaches:

1. **Multi-Instance Browser Testing**: Run multiple browser tabs or windows on development machine
2. **Cross-Browser Testing**: Test on Chrome, Firefox, and Edge
3. **Automated Test Scenarios**: Scripts to simulate player actions
4. **Network Simulation**: Tools to introduce latency and packet loss

Limitations:

- Automated testing limited for interactive gameplay
- Difficult to reproduce exact timing of race conditions
- Real-world network conditions vary widely

7.6.2 Lack of Automated Testing

Challenge:

Godot's testing ecosystem is less mature than traditional software development frameworks:

- No built-in unit testing framework
- GUI testing particularly difficult
- Networking code hard to test in isolation

Approach Taken:

- Manual testing with documented test cases
- Integration tests for critical paths
- Regression testing checklist before releases
- Community feedback as informal usability testing

Future Improvement:

- Integrate GUT (Godot Unit Test) framework
- Develop automated test suite for core logic
- Implement continuous integration pipeline

7.7 Educational Design Challenges

7.7.1 Balancing Accuracy and Playability

Challenge:

Educational games must balance pedagogical accuracy with engaging gameplay. Too much realism can reduce fun; too much simplification can create misconceptions.

Trade-offs Made:

- **Simplified Memory Model**: Actual shared-memory systems have complex cache hierarchies not represented
- **Abstracted Synchronization**: Real OpenMP requires explicit barriers, locks, and atomic operations
- **Idealized Performance**: Network latency doesn't perfectly model memory access patterns

Mitigation:

- Clear framing: "This is a metaphor, not a simulation"
- Post-game discussions to connect game to real concepts
- Accompanying documentation explaining simplifications
- Progressive difficulty introducing more complexity

7.7.2 Assessment of Learning Effectiveness

Challenge:

Demonstrating that the game actually improves learning outcomes requires formal educational research, including:

- Pre/post-test measurements
- Control groups using traditional instruction
- Statistical analysis of results
- Long-term retention studies

Limitations: Within the scope of this thesis, full educational efficacy studies were not feasible due to:

- Time constraints
- Need for institutional review board approval
- Difficulty recruiting sufficient participants
- Complexity of isolating game's effect from other factors

Future Work: Formal educational evaluation remains an important direction for future research, as discussed in Chapter 8.

7.8 Lessons Learned

7.8.1 Technical Lessons

1. **Framework Evaluation**: Thoroughly evaluate third-party frameworks before committing; test advanced use cases early.
2. **Abstraction Layers**: Isolate framework-specific code to facilitate future replacement if needed.
3. **State Management**: Design clear state management patterns early; retrofitting is expensive.
4. **Logging Infrastructure**: Comprehensive logging is essential for multiplayer debugging.
5. **Performance Profiling**: Profile early and often; don't assume where bottlenecks are.

7.8.2 Design Lessons

1. **User Testing**: Conduct user testing early; designer assumptions often wrong.
2. **Responsive Design**: Design for varying viewport sizes and input methods from the start, not as an afterthought.
3. **Simplicity**: Simpler interactions work better across input methods; complexity reduces usability.
4. **Feedback**: Clear, immediate feedback is critical for learning and engagement.

7.8.3 Process Lessons

1. **Iterative Development**: Iterative, incremental development surfaced issues early.
2. **Version Control**: Frequent commits with clear messages saved time during debugging.
3. **Documentation**: Documenting problems and solutions as they occurred proved invaluable.
4. **Community Engagement**: Engaging with open-source communities provided solutions and support.

7.9 Summary

This chapter documented the significant challenges encountered during development:

- **Technology Challenges**: Engine trade-offs, framework limitations, and performance concerns
- **GDSync Issues**: Protected mode blocking, incomplete documentation, and debugging difficulty
- **Synchronization Complexity**: Card ordering, race conditions, and selective visibility
- **UI/UX**: Screen space constraints, touch/mouse interaction precision, and device performance variability
- **Testing Difficulty**: Multiplayer testing complexity and lack of automated testing tools
- **Educational Design**: Balancing accuracy with playability and assessing learning effectiveness

Importantly, this chapter also presented solutions and lessons learned that will benefit future developers of similar educational multiplayer games. The next chapter concludes the thesis and discusses future directions for research and development.

Chapter 8

Conclusion and Future Work

This final chapter summarizes the contributions of this thesis, reflects on how the research questions were answered, discusses implications for HPC education and serious game development, and outlines comprehensive directions for future work.

8.1 Summary of Contributions

This thesis presented the design, implementation, and evaluation of the HPC Sorting Serious Game, a web-first educational tool for teaching parallel computing concepts through interactive card-sorting gameplay.

8.1.1 Primary Contributions

The main contributions of this work include:

1. Novel Pedagogical Approach

- Digital transformation of Professor D'Agostino's successful physical classroom experiments into a scalable, accessible web-based application
- Systematic mapping of HPC concepts (specifically OpenMP shared-memory parallelism and sequential vs. parallel execution) to concrete, manipulable game mechanics
- Demonstration that serious games can serve as effective icebreakers for teaching abstract parallel computing concepts

2. Technical Implementation

- Functional web-first serious-game-based educational tool demonstrating real-time multiplayer synchronization

- Solutions to multiplayer state management challenges, particularly selective visibility and host-authoritative architecture
- Integration of GDSync framework with an HTTP/SSE relay networking approach for browser-compatible multiplayer
- UI/UX design patterns for displaying and manipulating numerous interactive elements across varying viewport sizes and input methods

3. Documentation and Knowledge Transfer

- Comprehensive documentation of technical challenges (especially GDSync framework issues) and solutions
- Identification of design patterns and best practices for educational multiplayer game development
- Open-source codebase (MIT license) enabling reuse, extension, and further research
- Contributions to GDSync project documentation and issue resolution

4. Qualitative Validation

- Demonstration of technical feasibility through performance metrics and compatibility testing
- Positive preliminary usability feedback from informal user testing
- Evidence suggesting educational value as supplementary/introductory tool for HPC education
- Methodical tool and framework selection, documented in Chapter 3, improved implementation reliability and reduced rework risk

8.2 Research Questions Revisited

Returning to the research problem stated in Chapter 1:

> *How can we design and implement an effective serious game for web platforms that teaches fundamental High Performance Computing concepts (specifically OpenMP shared-memory parallelism and sequential vs. parallel execution) through interactive card-sorting gameplay while overcoming the technical challenges of multiplayer state synchronization and responsive UI/UX constraints?*

8.2.1 Addressing the Educational Problem

Q: How can abstract parallel computing concepts be mapped to concrete, understandable game mechanics?

A: The thesis demonstrated successful mapping through:

- Card-sorting as metaphor for data manipulation
- Shared container representing shared memory (OpenMP)
- local buffers representing thread-local storage (OpenMP)
- Card movement between shared container and local buffers representing memory access patterns
- Network latency representing synchronization and communication overhead
- Timer and move counter representing performance metrics

This mapping was validated through recognition by HPC-experienced participants and engagement by HPC-novice participants.

8.2.2 Addressing the Technical Problem

Q: How can we implement real-time multiplayer interaction in a web-first educational tool with acceptable performance?

A: The thesis demonstrated that:

- Browser-compatible HTTP/SSE relay networking enabled robust multiplayer synchronization in web deployment
- GDSync framework (despite challenges) simplifies state synchronization
- Host-authoritative architecture balances consistency and responsiveness
- Godot Engine provides adequate performance for 2D card-based gameplay

8.2.3 Addressing the Design Problem

Q: How can we balance educational effectiveness with engagement and playability in a web-first environment?

A: The thesis indicates that:

- Touch-based drag-and-drop provides intuitive, engaging interaction
- Visual clarity and immediate feedback support learning

- Scalable difficulty accommodates learners at different levels
- Game mechanics naturally lead to questions about parallel computing
- Informal testing suggests good engagement while supporting introductory educational goals

However, the design can be improved with better onboarding and educational scaffolding for learners without prior HPC exposure.

8.3 Implications

8.3.1 For HPC Education

This work has several implications for teaching parallel computing:

Complementary Tool, Not Replacement: Serious games are most valuable as **complementary tools** alongside traditional instruction:

- Use as pre-lecture icebreaker to introduce concepts
- Employ as reinforcement after formal instruction
- Leverage to uncover student misconceptions
- Deploy as discussion starter for deeper exploration

Accessibility Matters: Web-first design significantly enhances accessibility:

- No specialized HPC hardware required
- No app installation needed—runs directly in browsers
- Students can access from any device with a web browser
- Cross-platform compatibility (Windows, macOS, Linux, mobile)
- Reduces barriers to entry for HPC education

Visualization Is Powerful: Making abstract concepts visible and manipulable:

- Helps students build intuitive mental models
- Facilitates understanding before mathematical formalization
- Engages different learning styles
- Makes HPC concepts less intimidating

8.3.2 For Serious Game Development

This work offers insights for developers of educational multiplayer games:

Framework Selection is Critical:

- Thoroughly evaluate frameworks before committing
- Test advanced use cases early in development
- Build abstraction layers to facilitate future framework changes
- Engage with framework maintainers proactively

Multiplayer Adds Complexity:

- Start with single-player; add multiplayer incrementally
- Host-authoritative model simplifies consistency management for educational games
- Debugging multiplayer requires specialized tools and strategies
- Network variability must be designed for, not retrofitted

Browser/Platform Constraints Are Real:

- Design for varying viewports and input methods (mouse, touch) from the start
- Performance optimization is essential, especially for WebAssembly targets
- Browser security policies (mixed content blocking, Private Network Access) require careful deployment configuration
- Test on diverse browsers and hardware, not just development machines

8.4 Future Work

Numerous opportunities exist for extending and improving this work. Figure 8.1 presents a phased roadmap with dependency links between items.

8.4.1 Short-Term Enhancements

Improvements that could be implemented in 1–3 months:

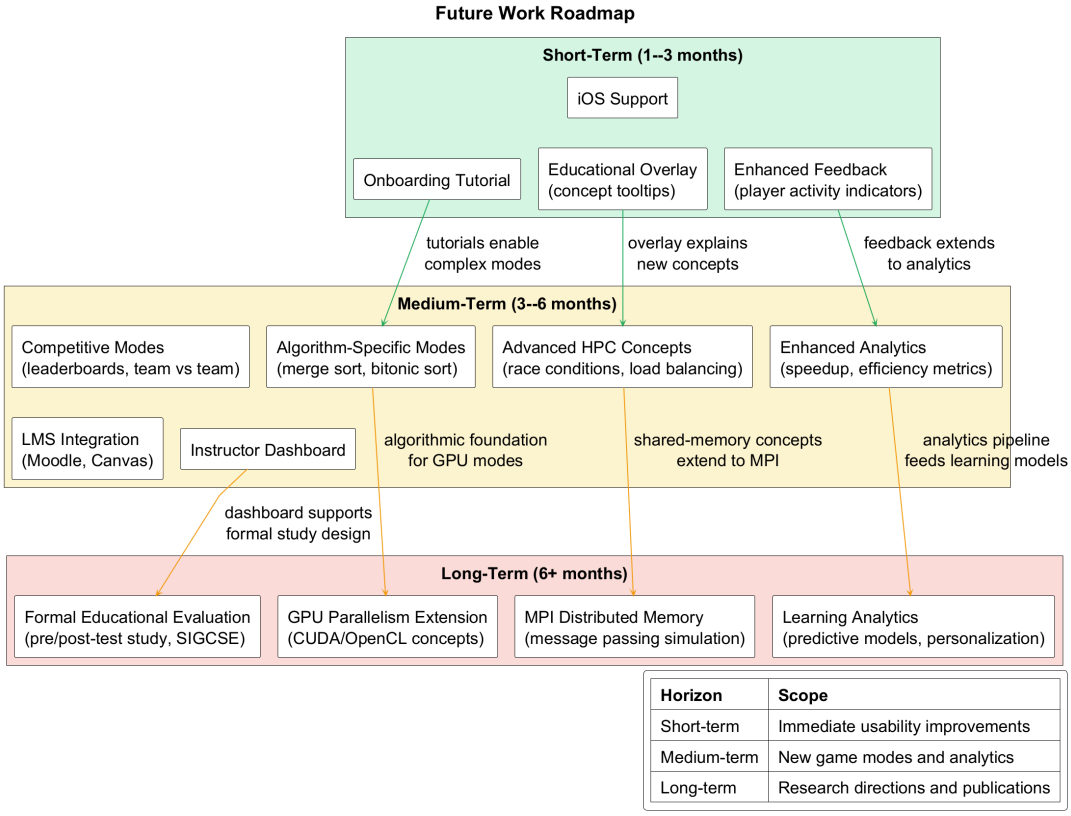


Figure 8.1: Future work roadmap organized by time horizon, with dependencies between items.

Onboarding and Tutorial:

- Interactive tutorial explaining game mechanics
- Tooltips and contextual hints for first-time users
- Guided first playthrough highlighting key concepts

Educational Overlay:

- Optional explanations connecting gameplay to HPC concepts
- Pop-up definitions of terms (thread, process, synchronization)
- Post-game summary explaining what was learned
- Links to additional resources for deeper learning

Enhanced Feedback:

- Player activity indicators showing what others are doing
- Visual representation of communication patterns
- Performance analysis comparing strategies
- Suggestions for improvement

iOS Support:

- Export to iOS using Godot's iOS templates
- Address Apple App Store requirements
- Test on iPhone and iPad devices

8.4.2 Medium-Term Extensions

Enhancements requiring 3–6 months of development:

Additional Game Modes:

1. **Guided Algorithm Modes**: The tool currently supports static partitioning strategies such as parallel merge sort through its existing mechanics (Chapter 4, Section 4.9). Future work could add *guided* modes that walk students through specific algorithms step-by-step [2]:

   - **Guided Parallel Merge Sort**: System-enforced phase progression (divide → local sort → merge rounds) with automatic partition assignment and barrier triggers, adding structure to the strategy players can already execute freely.
   - **Guided Parallel Quicksort**: This would require *new game mechanics* not present in the current tool: a pivot selection UI, partition validation, recursive sub-round management, and sub-group barrier synchronization. This represents a significant architectural extension rather than a guided overlay on existing mechanics.
   - Sample sort (distributed-memory style, future MPI extension)
   - Bitonic sort

   Table 8.1 summarizes the pedagogical distinctions and implementation effort.

Table 8.1: Potential guided-mode enhancements for parallel sorting strategies

Aspect	**Merge Sort**	**Quicksort**
Current support	Playable with existing mechanics	**Not supported**—requires new mechanics
Guided mode adds	Auto partition assignment, phase enforcement	Pivot UI, partition validation, recursive sub-rounds, sub-group barriers
Primary lesson	Barrier sync, reduction	Load imbalance, task fork
Effort estimate	Medium (3–4 weeks)	High (5–7 weeks)

2. **Advanced Concepts**: Introduce more HPC topics

   - Synchronization barriers and critical sections
   - Race conditions and deadlock scenarios
   - Load balancing challenges
   - Communication patterns (broadcast, scatter/gather, reduce)

3. **Competitive Modes**: Enhance engagement

   - Time trials with leaderboards
   - Team vs. team competitions
   - Challenge modes with specific constraints

Enhanced Analytics:

- Detailed performance metrics (speedup, efficiency, scalability)
- Strategy analysis and optimization suggestions
- Comparison with theoretical optimal performance
- Progress tracking across multiple sessions

Web Distribution Enhancements:

- HTML5 export for browser-based play
- Integration with learning management systems (Moodle, Canvas)
- Embeddable widget for course websites
- Cross-platform multiplayer (mobile + web)

Instructor Dashboard:

- Classroom management interface
- Real-time monitoring of student gameplay
- Assessment and analytics integration
- Customizable game parameters and scenarios

8.4.3 Long-Term Research Directions

Research directions requiring 6+ months and potential collaborations:

Formal Educational Evaluation: Conduct rigorous educational research to assess learning outcomes:

- **Study Design**: Pre/post-test controlled study with treatment and control groups
- **Participants**: Undergraduate students in HPC/parallel computing courses
- **Measures**: Knowledge assessments, concept inventories, attitude surveys
- **Analysis**: Statistical comparison of learning gains between groups

Learning Analytics Integration:

- Instrument game to collect detailed interaction data
- Analyze gameplay patterns to identify learning difficulties
- Develop predictive models for student understanding
- Personalize game difficulty based on demonstrated mastery
- Research publication in learning analytics venues (LAK, EDM)

GPU Parallelism Extension: Extend game to cover GPU computing concepts (CUDA, OpenCL):

- Massive parallelism simulation (thousands of "mini-workers")
- Warp/wavefront concept representation
- Memory hierarchy visualization (global, shared, local)
- Divergence and bank conflict demonstrations

Distributed Memory Extension (MPI): Future extension to simulate MPI distributed-memory programming:

- Separate player "nodes" with isolated memory spaces
- Explicit message passing between nodes (simulating MPI_Send/MPI_Recv)
- Support for hybrid MPI+OpenMP programming models
- Hierarchical parallelism with shared memory within nodes
- Load balancing across heterogeneous resources

AI-Assisted Learning: Integrate AI to enhance educational effectiveness:

- AI opponent for single-player mode
- Intelligent tutoring system providing hints
- Automatic difficulty adjustment
- Natural language explanations of concepts

8.4.4 Community and Ecosystem Development

Open Source Community Building:

- Publish to GitHub with comprehensive documentation
- Create contributor guidelines and issue templates
- Engage educational computing communities
- Accept contributions from students and educators
- Maintain active development and support

Educational Resource Development:

- Instructor guides with lesson plans
- Student worksheets and reflection prompts
- Video tutorials and gameplay examples
- Integration guides for popular LMS platforms
- Curriculum modules incorporating the game

Conference and Journal Publications: Disseminate findings to relevant research communities:

- **CS Education**: SIGCSE, ICER, ITiCSE
- **HPC Education**: EduHPC, XSEDE Education Track
- **Serious Games**: Games+Learning+Society, FDG
- **Educational Technology**: Learning @ Scale, EC-TEL
- **HPC Venues**: SC Education Program, ISC tutorials

Potential paper topics:

1. Design and implementation of a web-based serious game for HPC education
2. Evaluation of learning outcomes with serious game intervention
3. Technical challenges and solutions in educational multiplayer games
4. Pedagogical mapping from physical to digital activities

8.5 Limitations and Reflections

8.5.1 Methodological Limitations

8.5.1.1 Evaluation Scope:

- Informal usability testing with small sample (n=12)
- No formal educational efficacy study
- Limited diversity in participant demographics
- Short-term evaluation only (no long-term retention studies)

8.5.1.2 Technical Scope:

- Scope centered on web-first browser deployment; native app packaging was not a primary implementation target
- Focus on basic parallel concepts (no advanced topics)
- Simplified metaphors (not high-fidelity simulations)
- Limited integration with formal educational contexts

8.5.2 Reflections on the Development Process

What Worked Well:

- Iterative development allowed course correction
- Open-source toolchain reduced costs and increased flexibility
- Community engagement provided valuable support
- Documentation-first approach facilitated thesis writing
- Informal testing surfaced usability issues early

What Could Be Improved:

- Earlier, more frequent testing with target users
- More thorough framework evaluation before commitment
- Formal project management and milestone tracking
- Automated testing infrastructure from the start
- More structured approach to educational validation

8.6 Concluding Remarks

This thesis presented the design, implementation, and evaluation of the HPC Sorting Serious Game, demonstrating that:

1. **Serious-game-based tools can support HPC concept learning** through intuitive mechanics that make abstract ideas concrete and manipulable.
2. **Web-first platforms are viable for educational multiplayer tools**, despite technical challenges in synchronization, UI/UX design, and performance optimization.
3. **Open-source technologies enable accessible HPC education** by eliminating cost barriers and empowering educators to customize and extend educational tools.
4. **Interdisciplinary challenges inspire innovative solutions** at the intersection of computer science education, game design, parallel computing, and web deployment.

The HPC Sorting Serious Game represents a step toward making High Performance Computing education more accessible, engaging, and effective. By leveraging the ubiquity of web browsers and the motivational power of games, this work contributes to democratizing access to advanced computing concepts.

As parallel computing becomes increasingly fundamental to modern software engineering—from multicore CPUs to cloud computing to AI/ML workloads—innovative educational approaches are essential to prepare the next generation of computing professionals. Serious games, when thoughtfully designed and rigorously evaluated, can play a valuable role in this educational mission.

It is hoped that this work inspires further research and development in educational technologies for HPC and serves as a useful resource for educators, students, and game developers interested in the serious games domain.

8.7 Final Thoughts

Teaching parallel computing remains challenging, but it need not be boring or inaccessible. By combining pedagogical insight, technical innovation, and game design principles, it is possible to create learning experiences that are both effective and enjoyable.

The journey from physical card-sorting experiments in a classroom to a digital multiplayer web-based educational tool demonstrates the potential for technology to scale and extend proven teaching methods. While the game developed in this thesis is far from perfect, it represents a meaningful contribution to the ongoing effort to make HPC education more engaging and accessible.

Looking ahead, the convergence of mobile computing, cloud infrastructure, and educational research offers exciting possibilities for serious games in computing education. The potential for this work to evolve, inspire further innovations, and ultimately contribute to better outcomes for students learning parallel computing remains a motivating prospect.

"People overestimate what they can do in a year and underestimate what they can do in ten years."
— Bill Gates

Bibliography

[1] J. Mullen, L. Milechin, and D. Milechin, "Teaching and learning hpc through serious games," *Journal of Parallel and Distributed Computing*, vol. 158, pp. 115–125, 2021.

[2] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer, 2019, Chapter 5: Sorting and Selection, ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0

[3] M. Zyda, "From visual simulation to virtual reality to games," *IEEE Computer*, vol. 38, no. 9, pp. 25–32, 2005. DOI: 10.1109/MC.2005.297

[4] J. Piaget, *Genetic Epistemology*. New York: Columbia University Press, 1970.

[5] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. New York: Harper & Row, 1990.

[6] D. A. Kolb, *Experiential Learning: Experience as the Source of Learning and Development*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[7] T. M. Connolly, E. A. Boyle, E. MacArthur, T. Hainey, and J. M. Boyle, "A systematic literature review of empirical evidence on computer games and serious games," *Computers & Education*, vol. 59, no. 2, pp. 661–686, 2012.

[8] M. Papastergiou, "Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation," *Computers & Education*, vol. 52, no. 1, pp. 1–12, 2009. DOI: 10.1016/j.compedu.2008.06.004

[9] CodeCombat Inc., *Codecombat – coding games to learn python and javascript*, Accessed: 2026-02-19, 2026. [Online]. Available: https://codecombat.com/

[10] Lightbot Inc., *Lightbot – programming puzzles*, Accessed: 2026-02-19, 2026. [Online]. Available: https://lightbot.lu/

[11] Unity Technologies, *Unity real-time development platform*, Accessed: 2026-02-23, 2026. [Online]. Available: https://unity.com/

[12] Unity Technologies, *Unity asset store*, Accessed: 2026-02-23, 2026. [Online]. Available: https://assetstore.unity.com/

[13] Unity Technologies, *Unity manual: Webgl*, Accessed: 2026-02-23, 2026. [Online]. Available: https://docs.unity3d.com/Manual/webgl.html

[14] Unity Technologies, *Unity plans and pricing*, Accessed: 2026-02-23, 2026. [Online]. Available: https://unity.com/products/pricing-updates

[15] Epic Games, *Unreal engine*, Accessed: 2026-02-23, 2026. [Online]. Available: https://www.unrealengine.com/

[16] Epic Games, *Unreal engine documentation: Pixel streaming*, Accessed: 2026-02-23, 2026. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/pixel-streaming-in-unreal-engine

[17] Godot Community, *Godot engine: Free and open source 2d and 3d game engine*, Accessed: 2024, 2024. [Online]. Available: https://godotengine.org/

[18] Godot Engine contributors, *Godot engine license (mit)*, Accessed: 2026-02-23, 2026. [Online]. Available: https://github.com/godotengine/godot/blob/master/LICENSE.txt

[19] Godot Engine contributors, *Godot documentation: Exporting for the web*, Accessed: 2026-02-23, 2026. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_web.html

[20] Cocos, *Cocos creator*, Accessed: 2026-02-23, 2026. [Online]. Available: https://www.cocos.com/en/creator

[21] Cocos, *Cocos creator manual: Build and publish to web*, Accessed: 2026-02-23, 2026. [Online]. Available: https://docs.cocos.com/creator/manual/en/editor/publish/web-mobile.html

[22] Intel Corporation, *Intel vtune profiler*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[23] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[24] Argonne National Laboratory, *Mpich – high-performance portable mpi*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.mpich.org/

[25] The Open MPI Project, *Open MPI: Open source high performance computing*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.open-mpi.org/

[26] Tomorrow Corporation, *Human resource machine*, Accessed: 2026-02-19, 2015. [Online]. Available: https://tomorrowcorporation.com/humanresourcemachine

[27] Zachtronics, *Shenzhen i/o*, Accessed: 2026-02-19, 2016. [Online]. Available: https://www.zachtronics.com/shenzhen-io/

[28] T. Bell, I. H. Witten, and M. Fellows, *Cs unplugged – computer science without a computer*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.csunplugged.org/

[29] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[30] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[31] Emscripten Contributors, *Emscripten – an llvm-to-webassembly compiler*, Accessed: 2026-02-24, 2026. [Online]. Available: https://emscripten.org/

[32] GD-Sync Team, *Gd-sync: Multiplayer synchronization framework for godot*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.gd-sync.com/documentation

[33] HimaJyun, *PackRTC: WebRTC multiplayer addon for Godot engine*, Accessed: 2026-03-09, 2024. [Online]. Available: https://github.com/maji-git/packrtc

[34] S. Zinkowicz, *Gd-sync fork used in this thesis (web export fix branch)*, Accessed: 2026-02-19, 2026. [Online]. Available: https://github.com/Siponek/GD-Sync-fork

[35] Python Software Foundation, *Python programming language – official website*, Accessed: 2026-02-19, 2026. [Online]. Available: https://www.python.org/

[36] aio-libs contributors, *Aiohttp documentation*, Accessed: 2026-02-19, 2026. [Online]. Available: https://docs.aiohttp.org/

[37] Godot Asset Library, *Toast party (godot asset library)*, Accessed: 2026-02-19, 2026. [Online]. Available: https://godotassetlibrary.com/asset/PcSgaA/toast-party

[38] Godot Asset Library, *Vartree (godot asset library)*, Accessed: 2026-02-19, 2026. [Online]. Available: https://godotassetlibrary.com/asset/PUudIL/vartree

[39] Astral Software Inc., *Ruff documentation*, Accessed: 2026-02-19, 2026. [Online]. Available: https://docs.astral.sh/ruff/

[40] C. Casey, *Just programmer's manual*, Accessed: 2026-02-19, 2026. [Online]. Available: https://just.systems/

[41] F. FiloSottile, *Mkcert: Valid https certificates for localhost*, Accessed: 2026-02-19, 2026. [Online]. Available: https://github.com/FiloSottile/mkcert

[42] Chocolatey Software, Inc., *Chocolatey software – package manager for windows*, Accessed: 2026-02-19, 2026. [Online]. Available: https://chocolatey.org/

[43] game-gems, *Godot-const-generator*, GitHub repository. Accessed: 2026-02-24, 2026. [Online]. Available: https://github.com/game-gems/godot-const-generator

[44] The Git Project, *Git – distributed version control system*, Accessed: 2026-02-19, 2026. [Online]. Available: https://git-scm.com/

[45] GitHub, Inc., *Github platform*, Accessed: 2026-02-19, 2026. [Online]. Available: https://github.com/

[46] Szink, *Hpc sorting serious game: Open source repository*, Accessed: 2024, 2024. [Online]. Available: https://github.com/Siponek/hpc-sorting-serious-game

Appendix A

User Manual

A.1 Introduction

This user manual provides instructions for accessing, configuring, and playing the HPC Sorting Serious Game. It is intended for students, educators, and anyone interested in learning about parallel computing through interactive gameplay.

The game is deployed as a **web application** (Godot Engine WebAssembly export) and runs directly in a modern browser. No installation is required for the primary deployment. Desktop (Windows) builds are also available for offline use or development.

A.2 System Requirements

A.2.1 Web Version (Primary)

- **Browser**: Chromium-based browser (Chrome, Edge, Brave) or Firefox, version 145 or later
- **WebGL**: WebGL 2.0 support required
- **JavaScript**: Must be enabled
- **RAM**: 2 GB minimum available to browser
- **Network**: Internet connection required for multiplayer mode; single-player works offline after initial load
- **Input**: Mouse or touchscreen

A.3 Accessing the Game

A.3.1 Web Version

1. Navigate to the hosted game URL in a supported browser
2. Wait for the WebAssembly module to load (progress bar shown)
3. The main menu appears once loading is complete

Note: The web export is built with thread support disabled (`variant/thread_support=false` in the Godot export preset), so COOP/COEP headers and `SharedArrayBuffer` are *not* required. The game runs as a single-threaded WebAssembly application. Multiplayer communication uses HTTP/SSE, which does not depend on shared memory or cross-origin isolation.

A.4 Getting Started

A.4.1 Main Menu

On launch, the main menu presents the following options:

- **Single Player**: Practice sorting alone (sequential execution baseline)
- **Multiplayer**: Join or host a cooperative multiplayer game (OpenMP shared-memory simulation)
- **Options**: Configure theme and display options

A.5 Single-Player Mode

A.5.1 Starting a Single-Player Game

1. Select "Single Player" from the main menu
2. Configure game parameters in the options dialog:
    - **Number of cards**: configurable via spin box
    - **Buffer slot count**: number of local buffer positions
    - **Card value range**: upper bound for card values
3. Click "Start" to begin

A.5.2 Gameplay

Objective: Sort all cards in ascending order in the main container.

Controls:

- **Drag and Drop**: Click and drag a card to reposition it within the container or move it to/from a buffer slot
- **Scroll**: Use the horizontal scroll bar or mouse wheel to navigate through cards when they exceed viewport width

Using Buffer Slots:

- Buffer slots appear below the main card container
- Drag cards into buffer slots for temporary storage during sorting
- Buffer slots simulate thread-local storage in parallel computing
- Cards in buffers must form a contiguous subsequence of the sorted target before the game can complete

Winning:

- The game completes when all cards are in correct ascending order in the main container
- Completion time is displayed

A.6 Multiplayer Mode

A.6.1 Creating a Game

1. Select “Multiplayer” from the main menu
2. Select “Create Game”
3. Configure game settings:
    - Number of cards
    - Buffer slot count per player
    - Barrier mode (first-to-reach or round-robin)
4. A lobby code is generated automatically

5. Share this code with other players
6. Wait for players to join the lobby
7. Once all players are present, click “Start Game”

A.6.2 Joining a Game

1. Select “Multiplayer” from the main menu
2. Select “Join Game”
3. Enter the lobby code provided by the host
4. Click “Join”
5. Wait in the lobby for the host to start the game

A.6.3 Multiplayer Gameplay

Shared Container (Shared Memory):

- All cards are visible in the shared container, simulating shared memory accessible by all threads
- Any player can drag cards within the shared container to reorder them
- Changes propagate to all connected players in real time

Local Buffers (Partitioned Shared Memory):

- Each player has local buffer slots visible only to themselves
- Cards moved to a player’s buffer disappear from other players’ views
- When a card is returned to the shared container, it becomes visible to all again
- This simulates the distinction between shared and private memory in OpenMP

Barrier Synchronization:

- Any player can initiate a barrier (analogous to `#pragma omp barrier`)
- All players must reach the barrier before proceeding
- Once all players are at the barrier, a designated “main thread” (one player) can perform coordinated operations while others are blocked

- Main thread selection is either first-to-reach or round-robin, depending on game settings
- This teaches students about synchronization primitives in parallel computing

Winning Together:

- The team wins when all cards are sorted correctly in the shared container
- All players must have returned their buffer cards to the container

A.7 Settings

- **Theme**: Toggle between light and dark mode
- **Signaling Server URL**: Configure a custom signaling server address for multiplayer (advanced; defaults to the hosted server)

A.8 Tips and Strategies

A.8.1 Sorting Strategies

1. **Divide and Conquer**: Use buffer slots to sort smaller subsets of cards, then merge them back
2. **Find Sorted Runs**: Identify cards that are already in order and build around them
3. **Minimize Moves**: Think before moving—unnecessary rearrangements waste time

A.8.2 Multiplayer Coordination

1. **Divide Responsibility**: Each player takes a value range (e.g., player 1 handles cards 1–25, player 2 handles 26–50)
2. **Use Barriers Strategically**: Initiate a barrier when all players have sorted their portions and need to merge
3. **Communicate Externally**: Use voice chat or messaging to coordinate since the game does not include built-in chat
4. **Main Thread Awareness**: When you are the main thread during a barrier, merge sorted chunks efficiently; other players must wait

A.9 Troubleshooting

A.9.1 Common Issues and Solutions

Game does not load in browser:

- Verify that your browser supports WebGL 2.0 (check `chrome://gpu` in Chrome)
- Verify that JavaScript is enabled and WebGL 2.0 is supported (check `chrome://gpu` in Chrome)
- Try clearing browser cache and reloading
- Try a different Chromium-based browser

Performance issues:

- Reduce the number of cards
- Close other browser tabs to free memory
- Ensure hardware acceleration is enabled in browser settings

Cannot connect to multiplayer:

- Verify internet connection is active
- Check that both players are using the same game version
- Ensure the signaling server is reachable (default server requires internet access)
- Re-enter the lobby code carefully
- If using a custom signaling server, verify it is running and accessible

Multiplayer desynchronization:

- This can occur if network packets are lost or arrive out of order
- The host can use the "sync state" mechanism to push authoritative state to all clients
- As a last resort, restart the game session

A.10 Educational Use

A.10.1 For Students

- Play single-player mode first to understand the card sorting mechanics
- Observe how buffer slots help organize your sorting work (thread-local storage)
- In multiplayer, notice how coordination becomes harder with more players (communication overhead)
- Pay attention to the barrier mechanic—it directly mirrors `#pragma omp barrier`
- Reflect on how game concepts relate to OpenMP parallel programming

A.10.2 For Instructors

- Use as a pre-lecture activity to introduce HPC concepts before formal instruction
- The web deployment requires no installation—students can play immediately via a shared URL
- Facilitate post-gameplay discussion connecting game mechanics to parallel programming:
  - Shared container ↔ shared memory
  - local buffers ↔ thread-local / private variables
  - Barrier mechanic ↔ `#pragma omp barrier`
  - Multiple players ↔ multiple threads
  - Single-player ↔ sequential execution baseline
- Use completion metrics to discuss relative effort of parallel coordination vs. sequential sorting

A.11 Support and Resources

- **Game Repository**: https://github.com/Siponek/hpc-sorting-serious-game
- **Thesis Repository**: https://github.com/Siponek/HPC-thesis
- **Issue Tracker**: Report bugs and request features via GitHub Issues
- **License**: MIT license—contributions are welcome

Appendix B

Code Listings

B.1 Source Code Repository

The complete source code for the HPC Sorting Serious Game is available as an open-source project:

- **Game Repository**: https://github.com/Siponek/hpc-sorting-serious-game
- **GDSync Fork**: https://github.com/Siponek/GD-Sync-fork (web-export fix; upstream PR #108)
- **License**: MIT License
- **Language**: GDScript (Godot Engine 4.x)

B.2 Project Structure

The project follows Godot Engine conventions. The listing below shows the key directories and files relevant to this thesis:

```
hpc-sorting-serious-game/
+-- project.godot                 # Project configuration
+-- scene_manager.gd              # Global scene transition autoload
+-- settings.gd                   # Global settings autoload
+-- theme_manager.gd              # Dark/light theme autoload
+-- export_presets.cfg            # Export presets (Web, Windows, Android)
+-- scenes/
|   +-- CardScene/
|   |   +-- scripts/
|   |   |   +-- card_manager.gd  # Base card manager (705 lines)
|   |   +-- card_scene.tscn      # Card prefab scene
```

```
|   +-- MainMenuScene/
|   |   +-- menu_scene.tscn      # Main menu
|   |   +-- Singleplayer/
|   |   |   +-- singleplayer_options.gd
|   |   +-- Multiplayer/
|   |       +-- multiplayer_options.gd
|   +-- Singleplayer/
|   |   +-- singleplayer_scene.tscn
|   +-- Multiplayer/
|   |   +-- connection_manager.gd       # GDSync signal wiring (348 lines)
|   |   +-- MultiplayerGame/
|   |   |   +-- multiplayer_card_manager.gd  # Extends card_manager (1143 lines)
|   |   |   +-- barrier_manager.gd           # Barrier state machine (113 lines)
|   |   +-- Lobby/
|   |   |   +-- multiplayer_lobby.gd
|   |   +-- GDSyncWebPatch/
|   |       +-- gdsync_web_patch.gd          # Web platform patch (67 lines)
|   |       +-- local_server_signaling.gd    # HTTP-based LocalServer (~600 lines)
|   |       +-- signaling_client.gd          # HTTP/SSE signaling client
+-- addons/
|   +-- GD-Sync/                 # GDSync plugin (git submodule, forked)
|   +-- var_tree/                # Runtime variable inspector
|   +-- toastparty/             # Toast notification plugin
|   +-- scene-selector/         # Scene selector plugin
+-- lib/
|   +-- logger/                 # Structured logging library
|   +-- feature_flag_handlers/  # Feature flag utilities
+-- res/
|   +-- algorithms/             # Algorithm description resources
|   +-- dark_theme.tres         # Dark theme resource
|   +-- menu_theme.tres         # Menu theme resource
|   +-- shaders/                # Visual effect shaders
+-- signaling-server/
|   +-- server.py               # Python aiohttp signaling server
|   +-- server/                 # Server module package
|   +-- tests/                  # Server test suite
+-- exports/
|   +-- web-export/             # Web build output
|   +-- certs/                  # TLS certificates for local dev
```

B.3 Key Components

B.3.1 Card System

- `card_manager.gd`: Base class handling card generation, drag-and-drop, buffer slot management, sorting validation (`check_sorting_order()`), buffer contiguity checking, and swap animations

- `multiplayer_card_manager.gd`: Extends `card_manager.gd`; adds `CardState` serialization, GDSync function exposure, network broadcast/reconciliation, barrier integration, and visibility management for multi-player buffers

- `barrier_manager.gd`: Three-state machine (`RUNNING` → `WAITING_AT_BARRIER` → `BARRIER_ACTIVE`) implementing `#pragma omp barrier` semantics with main-thread selection

B.3.2 Networking and Multiplayer

- `connection_manager.gd`: Autoload that wires GDSync signals (`connected`, `lobby_created`, `client_joined`, etc.) to game-level events; maintains typed player map

- `gdsync_web_patch.gd`: Autoload that detects `OS.has_feature("web")` and replaces GDSync's native `LocalServer` with the HTTP/SSE-based `local_server_signaling`

- `local_server_signaling.gd`: HTTP-based replacement for GDSync's `LocalServer`; preserves the original API contract while using the signaling server for packet relay

- `signaling_client.gd`: HTTP/SSE client for lobby management and real-time event streaming

B.3.3 Infrastructure

- `scene_manager.gd`: Global autoload managing scene transitions with loading screen

- `settings.gd`: Global autoload storing game configuration (card count, buffer count, barrier mode, multiplayer flag)

- `theme_manager.gd`: Dark/light theme switching

- `server.py`: Python signaling server built on `aiohttp`; handles lobby create/join, SSE event streaming, and game packet relay

B.4 Configuration

B.4.1 Autoloads

The following singletons are registered in `project.godot` and available globally:

- **Settings**: Game parameters (card count, buffer count, barrier mode)
- **SceneManager**: Scene transitions
- **ThemeManager**: Visual theme control
- **ConnectionManager**: Multiplayer session state
- **GDSync**: Networking framework (third-party plugin)
- **GDSyncWebPatch**: Web-export compatibility layer
- **ToastParty**: User notification toasts

B.4.2 Web Export Settings

Key settings in the `export_presets.cfg` Web preset:

- **Export Format**: WebAssembly (WASM)
- **Thread Support**: Disabled (`variant/thread_support=false`); COOP/-COEP headers not required
- **Canvas Resize Policy**: Responsive
- **Output**: `exports/web-export/` directory

B.5 Building and Running

B.5.1 Development Setup

1. Install Godot Engine 4.x
2. Clone the repository with submodules: `git clone -recursive`
3. Open `project.godot` in the Godot Editor
4. Press F5 to run locally (desktop mode)

B.5.2 Web Export

1. In Godot Editor: Project → Export
2. Select the "Web" preset
3. Click "Export Project"
4. Deploy the contents of `exports/web-export/` to a web server that sets the required COOP/COEP headers

The project includes a `justfile` with automation targets for common tasks (formatting, linting, export, server management).

B.5.3 Signaling Server

The HTTP/SSE signaling server is in the `signaling-server/` directory:

1. Install Python 3.10+ and dependencies: `pip install aiohttp aiohttp-cors`
2. Run: `python server.py`
3. Default port: 8765 (configurable)

See Appendix C for the signaling server API reference.

Appendix C

API Documentation

C.1 Introduction

This appendix documents the APIs and interfaces of the HPC Sorting Serious Game, focusing on the signaling server REST/SSE endpoints and the key GDScript classes. The signaling server is the only custom server-side component; the game client is built in Godot Engine 4.x with GDScript.

C.2 Signaling Server API

The signaling server is a Python application built on `aiohttp`. It serves three route groups: HTTP session management, HTTP lobby management with SSE, and WebSocket signaling for WebRTC.

C.2.1 Session Endpoints

These endpoints manage rooms for WebRTC signaling.

C.2.2 HTTP Lobby Endpoints (SSE)

These endpoints replace WebSocket-based lobby handling for browser compatibility. Real-time events are delivered via Server-Sent Events (SSE).

C.2.3 SSE Event Types

The SSE stream (`GET /api/lobby/events`) delivers the following event types:

Table C.1: Session management endpoints

Method	Path	Description
POST	/session/host	Create a new room. Body: {channel, lobby_name, public, player_limit}. Returns {code, ws_url}.
POST	/session/join/{code}	Join a room by code or lobby name. Returns {code, ws_url}.
POST	/session/update/{code}	Update room metadata (lobby name, public flag, player limit).
POST	/session/players/{code}	Update player count for a room.
POST	/session/close/{code}	Close a room and its associated lobby.

Table C.2: HTTP lobby endpoints

Method	Path	Description
POST	/api/lobby/connect	Connect and receive a peer ID. Optional body: {client_id} to specify a GDSync client ID.
POST	/api/lobby/disconnect	Disconnect from the server.
POST	/api/lobby/create	Create a new lobby. Body: {lobby_name, public, player_limit}. Returns {code, lobby}.
POST	/api/lobby/join	Join a lobby. Body: {code} or {lobby_name}.
POST	/api/lobby/leave	Leave the current lobby.
GET	/api/lobby/list	List public lobbies.
POST	/api/lobby/broadcast	Broadcast a game packet to lobby peers. Body: {data}.
GET	/api/lobby/events	SSE event stream. Query: ?peer_id={id}.

Table C.3: SSE event types

Event	Description
`welcome`	Initial connection acknowledgement with peer ID
`peer_joined`	A new peer joined the lobby
`peer_left`	A peer left the lobby
`lobby_joined`	Confirmation that this peer joined a lobby
`lobby_closed`	The lobby was closed (with reason)
`game_packet`	A game data packet relayed from another peer
`heartbeat`	Keep-alive ping
`error`	Error notification
`server_shutdown`	Server is shutting down

C.2.4 WebSocket Endpoints

These provide WebRTC signaling (for native desktop clients) and legacy WebSocket lobby handling:

Table C.4: WebSocket endpoints

Path	Description
`/ws/{code}`	WebRTC signaling channel. Exchanges ICE candidates, SDP offers/answers between peers in a room.
`/lobby`	WebSocket-based lobby management (native client usage; browser clients use HTTP+SSE instead).

C.2.5 Utility Endpoints

Table C.5: Utility endpoints

Method	Path	Description
`GET`	`/health`	Health check. Returns room, lobby, and peer counts.
`GET`	`/rooms`	List all active rooms.
`GET`	`/lobbies`	List all lobbies with peer details.
`GET`	`/api/server/info`	Server information (version, uptime, configuration).
`GET`	`/`	Welcome page / root info.

C.2.6 Error Codes

All error responses follow the format `{"success": false, "error": "<code>", "message": "..."}`.

Table C.6: Signaling server error codes

Code	Meaning
`LOBBY_NOT_FOUND`	Specified lobby does not exist
`LOBBY_CLOSED`	Lobby has been closed
`LOBBY_FULL`	Lobby has reached player limit
`ALREADY_IN_LOBBY`	Peer is already in a lobby
`NOT_IN_LOBBY`	Peer is not in any lobby
`ROOM_NOT_FOUND`	Specified room does not exist
`PEER_NOT_FOUND`	Specified peer ID not found
`PEER_ID_IN_USE`	Requested peer ID is already taken
`UNKNOWN_COMMAND`	Unrecognized message type
`INVALID_JSON`	Malformed JSON in request body

C.3 Server Data Models

The signaling server uses two primary data models:

Peer: Represents a connected player. Fields: `peer_id` (int), `ws` (WebSocket connection, optional), `sse_queue` (asyncio Queue for SSE events, optional), `player_data` (dict), `lobby_code` (str, optional). A peer uses either WebSocket *or* SSE, not both—this dual-transport design enables both native and browser clients.

Lobby: Represents a game lobby. Fields: `code` (str), `name` (str), `host_id` (int), `public` (bool), `player_limit` (int), `open` (bool), `created_at` (ISO timestamp), `peers` (dict of Peer).

C.4 GDScript Game Classes

This section summarizes the key GDScript classes. Full source code is available in the repository (see Appendix B.1).

C.4.1 CardManager (card_manager.gd)

Base class managing all card logic for single-player mode.

Key Properties:

- `num_cards: int` — Number of cards in the game
- `cards_array: Array` — All card node instances
- `sorted_all: Array` — Sorted reference array (target ordering)
- `slots: Array` — Buffer slot instances
- `card_container: HBoxContainer` — Shared card container (scene tree node)

Key Methods:

- `check_sorting_order() -> bool` — Validates ascending order of cards in container
- `check_player_buffer_contiguity() -> bool` — Validates buffer cards form a contiguous sorted subsequence
- `generate_completed_card_array(values, prefix) -> Array` — Creates card instances from value array
- `create_buffer_slots() -> Array` — Instantiates player buffer slots
- `animate_card_swap(card_a, card_b)` — Animated card position swap with tween

C.4.2 MultiplayerCardManager (multiplayer_card_manager.gd)

Extends `CardManager` for multiplayer mode.

Inner Class — CardState:

- Serializable state object with `to_dict()` and `from_dict()` methods
- Fields: `value`, `index`, `original_index`, `in_container`, `in_buffer`, `buffer_owner`

Key Methods (GDSync-exposed):

- `sync_complete_game_state(state_data)` — Full state reconciliation
- `sync_card_moved(value, from_idx, to_idx, client_id)` — Card repositioning
- `sync_card_entered_buffer(value, player_id)` — Card enters local buffer

- `sync_card_left_buffer(value, player_id, new_idx)` — Card leaves buffer
- `barrier_thread_reached(player_id)` — Player reached barrier
- `barrier_activate()` — All threads at barrier; activate
- `barrier_release()` — Release barrier; resume normal play

C.4.3 BarrierManager (barrier_manager.gd)

Implements barrier synchronization state machine.

States: `RUNNING` → `WAITING_AT_BARRIER` → `BARRIER_ACTIVE` → `RUNNING`.

Key Methods:

- `enter_waiting_state(initiator_id, all_player_ids)` — Transition to waiting state
- `activate_barrier()` — Transition to active state (all threads arrived)
- `release_barrier()` — Reset and return to running state
- `determine_main_thread(first_id, all_ids) -> int` — Select coordinator (first-to-reach or round-robin)

Signals:

- `barrier_state_changed(new_state)` — State transition notification
- `thread_reached_barrier(player_id)` — Player arrival notification
- `main_thread_assigned(player_id)` — Main thread selected

C.4.4 ConnectionManager (connection_manager.gd)

Autoload that wraps GDSync signals into game-level events.

Key Methods:

- `am_i_host() -> bool` — Whether local player is lobby host
- `get_my_client_id() -> int` — Local player's GDSync client ID
- `get_player_list() -> MultiplayerTypes.PlayersMap` — Current player map

Key Signals:

- `player_joined_lobby(player_id)` — Player entered lobby
- `player_list_updated(players)` — Player list changed
- `lobby_joined(lobby_name)` — Local player joined a lobby

C.5 GDSync Framework Usage

This project uses GDSync's explicit function exposure model. GDSync does *not* use annotations or decorators for RPC. Instead, functions are registered at runtime:

```
# Expose a function for remote calls
GDSync.expose_func(self.my_function)

# Call function on all peers
GDSync.call_func(self.my_function, [arg1, arg2])

# Call function on a specific peer
GDSync.call_func_on(target_peer_id, self.my_function, [arg1, arg2])

# Expose a variable for synchronization
GDSync.expose_var(self, "variable_name")
```

Listing C.1: GDSync function exposure (actual API)

Key GDSync Signals:

- `GDSync.connected` — Successfully connected to server
- `GDSync.connection_failed` — Connection attempt failed
- `GDSync.lobby_created` — Lobby created (host)
- `GDSync.lobby_joined` — Joined an existing lobby
- `GDSync.client_joined` — A new client joined the lobby
- `GDSync.client_left` — A client left the lobby
- `GDSync.lobbies_received` — Lobby list received

For protected mode details and the web-export compatibility layer, see Chapter 7, Section 7.3.1 and Chapter 5, Section 5.6.